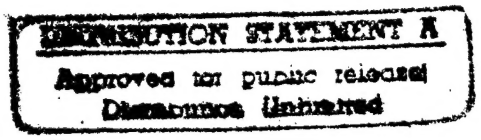
September 1985

Report No. STAN-CS-85-1069

The Logical Data Model: A New Approach To Database Logic

by

Gabriel Mark Kuper



Department of Computer Science

Stanford University
Stanford, CA 94305

THE LOGICAL DATA MODEL:

A NEW APPROACH

TO DATABASE LOGIC

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Gabriel Mark Kuper
September 1985

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jeffrey D. Ullman
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Christos H. Papadimitriou

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Moshe Y. Vardi
(CSLI)

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies & Research

Abstract

We propose a mathematical framework for unifying and generalizing the principal data models, i.e., the relational, hierarchical and network models. Until recently most theoretical work on databases has focused on the relational model, mainly due to its elegance and mathematical simplicity compared to the other models. Some of this work has pointed out various disadvantages of the relational model, among them its lack of semantics and the fact that it forces the data to have a flat structure that the real data does not always have.

The Logical Data Model (LDM) combines the advantages of the relational, network and hierarchical approaches. It models database schemas as directed graphs, in which the leaves correspond to the attributes, and the internal nodes to connections between the data. Instances of LDM schemas consist of r-values, which constitute the data space, and l-values, which constitute the address space. We are thus able to deal with instances of cyclic structures, but still get a first-order theory.

We define a logic on LDM schemas in which integrity constraints can be specified, and use it to define a logical, i.e., non-procedural, query language that is analogous to Codd's relational calculus. We also describe an algebraic, i.e., procedural, query language and prove that the two languages are equivalent. These languages have a novel feature: not only can they access a non-flat data structure, e.g. a hierarchy, but the answers they produce do not have to be flat either. Thus, the language really does have the ability to restructure data and not only to retrieve it, and can therefore be used both as a query language and for defining views.

Acknowledgments

I would like to especially thank my adviser, Jeff Ullman. He both suggested this area as a good one for research, and was very helpful in advising me which directions to explore. I would like to thank Moshe Vardi, Dave Maier, Christos Papadimitriou, Gio Wiederhold, Ernst Mayr and Richard Hull for comments on my work. Moshe Vardi in particular was very helpful in defining the logical query language. I would also like to thank my officemates Howard Trickey, Hank Korth, Jerry Plotnick, Eric Berglund, Joe Pallas, Vineet Singh, and Kai Yue.

This thesis was produced using LaTeX, a macro package designed by Leslie Lamport for Don Knuth's TeX typesetting system. The bibliography was prepared with BibTeX, written by Oren Patashnik. Financial assistance for this work was provided by AFOSR grant 80-0212, and NSF grant IST-12791.

Contents

List of Figures

Chapter 1

Introduction

This thesis proposes a new model for data, the *Logical Data Model* (LDM). The purpose of the LDM model is to combine the advantages of what are currently the principal data models. Most database systems are based on either a hierarchical or a network model [COD71] [ANS75] [IBM78] [Wie83] [Dat81] [Ull82], both of which describe in detail how the data is stored in the computer. Because of this, databases based on these models can be implemented efficiently, but on the other hand they are awkward to use, since the user has to be aware of a lot of details about the physical implementation.

For this reason, Codd [Cod70] introduced the relational model. In the relational model, the user's view of the data is that it is stored in tables, and he does not have to be aware of the precise details of the physical implementation. Codd [Cod72] defined two query languages on relational databases. One of these is a logical, i.e., non-procedural, language, which is used to specify what the result of the query should be, without describing explicitly how to compute it. The second language is an algebraic, i.e., procedural, language, equivalent to the logical language, which the system uses to answer the query. These query languages have a unique property not shared by network and hierarchical database management systems: The result of a query is a relation, i.e., has the same structure as the data in the original database. One consequence of this property is that the same language can be used for view definition, and another consequence is that the query language can handle complex queries by breaking them up into simpler subqueries.

The relational model introduces another level of abstraction between the physical representation of the data and what the user actually sees. As a result, they are harder to implement efficiently than network and hierarchical systems. The implementation problems have by now been solved, to a large extent [Tod76] [Zlo77] [SWKH76] [A*76]. Besides the issue of efficiency, however, the relational model has another disadvantage. By forcing the data to have a flat structure, i.e., by requiring that all the data be in the form of tables, some of the semantics of the data is lost [Cod70] [HM81] [SS75] [SS77a]. For example, if there is a natural connection in the data between individual objects and sets of objects of another type, we lose some of the structure of the data by forcing it into a first normal form relation [JS82]. While it is always possible in some way to encode the information in a relational form, this is not always the most natural thing to do. As another example, hierarchical and network database management systems have the ability to use virtual records. These are essentially pointers to physical records, and are used to avoid redundancy in the database [Ull82]. Update anomalies are one of the consequences of the fact that the relational model does not model virtual records.

The logical data model combines the advantages of both approaches. As in network and hierarchical databases, the data has more structure than in the relational model. In particular, we can use the LDM model to model cyclic structures and virtual records. On the other hand, we do not lose the advantages of the relational model. As in the relational model, our model has two query languages: A logical, i.e.,

non-procedural, and an equivalent algebraic, i.e., procedural, language. These languages are analogous to the relational calculus and algebra, and have the novel feature that not only can they access a non-flat data structure, e.g., a hierarchy, but the answers they produce do not have to be flat either. Thus, the language really does have the ability to restructure data and not only to retrieve it.

The organization of the thesis is as follows. Chapter 2 describes some related work. In Chapter 3, we give an informal description of the LDM model. We show how to map various data models into the logical data model, and give some informal examples of the two query languages. Chapter 4 contains the formal definitions of LDM schemas and instances.

In the following two chapters, we define the logical query language. In Chapter 5 we define a logic on LDM schemas. We prove various results about the logic, including the fact that it is equivalent to a certain first-order logic. We also give a proof theory for the logic, and some complexity results. In Chapter 6 we use the logic to define a logical query language. We also discuss when a logical query is safe, and conclude with some complexity results.

In Chapter 7 we define the algebraic query language, and show that it is equivalent to the logical language. In Chapter 8 we investigate the role of cyclicity in database schemas. We show that under one measure of information content, cycles are unnecessary, i.e., anything that can be represented by a cyclic schema can also be represented by some acyclic schema. We conclude, in Chapter 9 with some directions for future work.

Chapter 2

Previous Work

2.1. Database Logic

Jacobs [Jac79] [Jac80] [Jac82] defined what he called "database logic." Database logic is a mathematical model of databases that claims to generalize the relational, network and hierarchical models. In database logic, a database schema is a set of rules of the form $R_j = (R_{j_1}, \ldots, R_{j_k})$. An instance of such a schema is essentially a table, in which the entries can themselves be tables rather than simple attributes. His model is a natural way to describe a hierarchy, and it can also be used to describe a network. Jacobs then defines a logical query language on database schemas.

His model has various shortcomings. One, relatively minor, is that the representation of a hierarchy does not allow virtual records. A more serious problem is how he handles cyclicity. He allows schemas to contain cycles, but explicitly forbids cycles on the instance level. Besides this, he also has an unnecessarily complicated definition of nesting depth. The lack of cyclicity in instances is a severe restriction on the expressive power of the model.

Another shortcoming of his model is the definition of a database instance. Since instances are acyclic, he is able to construct instances bottom-up. The problem is that his definition is rather complicated, and as the users views of the data consists of precisely these instances, we would like them to be as simple as possible.

Finally, the logic is not first-order. While using a more powerful logic does increase the expressiveness of the logic, it also makes it harder to handle mathematically. In fact the query language turns out to be too powerful, as it enables one to write queries whose result is not computable [Var83]. This is one reason why he does not define an equivalent algebraic language, and therefore his model contains only a logical, i.e., nonprocedural, query language.

2.2. The Format Model

The "format model" was introduced by Hull and Yap [HY82]. The format model is an attempt to generalize the relational and hierarchical models. A database schema, or format, is a tree with labels. The leaves correspond to the attributes in the relational model, and the internal nodes represent various connections between the data.

More formally, formats are made from fundamental components, called *basic types*, and three constructors, *composition*, *collection* and *classification*. A format is a tree with labels assigned to the nodes: Basic

types are assigned to the leaves, and the other constructors are assigned to the internal nodes. The notation they use is: □ for basic types, ⌓ for composition, O for collection and △for classification.

Each basic type has a corresponding *domain*, i.e., a set of values. The domains of the internal nodes are defined as follows. The composition constructor, ⌓, is similar to the cartesian product in the relational model, and to the aggregation of [SS77a]. The domain of a node of type ⌓ is the cartesian product of the domains of its children. The second constructor is classification, △ that is similar to the generalization of [SS77b]. The domain of such a node is the marked union of the domains of its children. Finally collection, O, is used to specify formation of sets of objects, all of a given type. Such a node has only one child, and its domain is the set of all finite subsets of the domain of the child.

An instance of a schema consists of assigning to each leaf some subset of the corresponding domain, and to each internal node some subset of the domain that is derived by the above rules.

Their motivation for introducing the format model was different from ours. They wanted to investigate notions of relative information capacity of database schemas, i.e., whether one database schema is more expressive than another. For that reason, they did not define a query language on their model. We described their model here, since the logical data model is based on their structuring of data, with several modifications. In particular, we modified the format model to allow cyclic structures, and thus we obtained a model that is a true generalization of the network and hierarchical models.

2.3. Non-First-Normal Form Relations

The relational model of [Cod70] restricts the relations in the database to what are called first-normal form, or normalized, relations. In non-first-normal form the components of a tuple in a relation are simple, i.e. atomic, objects, without any further structure. Various people, among them Makinouchi [Mak77], Scheck and Pistor [SP82] and Kobayashi [Kob80] have pointed out that for some applications such as picture data processing and CAD restricting the components to atomic objects is too restrictive a requirement.

[Mak77] and [OY85] discuss how to extend dependency theory and normal forms to non-first-normal form relations. [JS82], [AB84] and [FK77] define algebras for such relations. One consequence of our work will be that besides generalizing their work, we also get a logical, non-procedural, query language for non-first-normal form relations.

2.4. Non-Procedural Query Languages for the Network Model

Various papers, among them [MP82], [Tsi76], [Dat80] and [Gra79], have advocated using high-level languages for network databases. The languages they describe are all procedural. [MP82] and [Tsi76] describe what is essentially a relational front end for a network DBMS. Date's model [Dat80] involves explicit navigation as in CODASYL, and [Gra79] describes some ideas for automatic navigation using "paths" but does not describe how to use them in a query language.

[Day79], [DB82] and [GDB82] describe NQUEL, a non-procedural language similar to QUEL for use with network databases. The result of an NQUEL query is a relation, but there is also an NQUEL view definition language that creates new networks. They obtained an equivalent procedural language by mapping the network database into an equivalent relational one [Bor78] [Kay75], and then using the standard relational theory. Our approach differs from theirs in several ways. One difference is that the logical data model can handle more general structures then NQUEL. Another difference is that by defining the query languages directly on the given database schema, rather than through mapping them into the relational model, we get a more natural query language.

2.5. Non-Procedural Query Languages for the Hierarchical Model

Hardgrave in [Har78] looks at ways to define a non-procedural query language on hierarchical databases. The principal idea is that of a "broom," i.e., a node together with all its children and ancestors. Brooms in his model play the role of tuples in the relational model. The main problem he investigates is how to handle conditions on the tuples. For example, if u, v and w are nodes in the hierarchy, and the query is

$$\text{Print } u \text{ where } v = c_1 \text{ and } w = c_2$$

do we mean all those u's that are in some broom with $v = c_1$ and $w = c_2$, or all those u's that are in some broom with $v = c_1$ and in some other broom with $w = c_2$? He shows that there are four different approaches that may be taken, each of which differs from the others for some queries. Furthermore, he claims that users with different backgrounds and experience may expect the system to behave according to different ones of these approaches. Our query language does not make any of these assumptions for the user, but can be used to specify explicitly any of Hardgrave's query languages.

2.6. Statistical Databases

Models that have been proposed for statistical databases such as SSDB [OO84] and GRASS [BRR82] [RR83] [RR84] require that the data have more structure than the relational model provides. The structuring of the data is similar to that of non-first-normal form relations or to the format model that we described above, together with special nodes for aggregation. We can describe the structuring of data in these models using the logical data model, and it should be possible to extend the LDM model to include aggregation operations.

Chapter 3

Introduction to the Logical Data Model

3.1. Data Structuring in the Logical Data Model

The logical data model is based on Hull and Yap's format model (see Section 2.2). A database schema in the format model is a labeled tree. Leaves are labeled with basic types (□) that correspond to attributes, while internal nodes, labeled ⌓, ○ and △, correspond to composition, collection and classification, respectively.

As we mentioned in Section 2.2, the format model fails to model an important part of network and hierarchical database systems, namely the ability to use virtual records. To model this, we have to introduce cyclicity into the database schemas. Our first idea was to have two types of leaves: Basic types and *pointer nodes*, i.e., nodes that point to other nodes in the tree. It turned out, however, that what we wanted to express using pointer nodes could be expressed more simply if we use directed graphs rather than trees for the underlying schema.

We made two further modifications to Hull and Yap's format model schemas, both relatively minor. We have only one basic type, rather than several different ones. For our purposes, the distinction between the domains of the attributes is not important for structuring the data. In order to keep the model as simple as possible, we prefer to have only one basic type. We can express the fact that the values of some attribute come from a specific domain by a constraint in the LDM logic that we shall define later. In contrast, since Hull and Yap were interested mainly in relative information capacity of different database schemas, the distinction between different basic types was very important for them.

The other modification we made to the format model was to use *multigraphs* rather than simple directed graphs. This means that there may be more than one edge between two nodes, and enables different components of tuples to have the same structure.

Since it is more intuitive, we shall continue to use tree terminology when referring to LDM schemas. In particular, by *leaf* we shall mean a sink, and by *children* we shall mean successors.

In short, an LDM schema is a labeled directed multigraph. The leaves are labeled □ (basic type). The values that an instance of such a node can have are elements of some fixed domain. These nodes are analogous to attributes in the relational model. Each interior node is labeled with one of the following.

1. Composition, written ⌓. The domain of such a node is the cartesian product of the domains of its children.

2. Collection, written ○. The domain of such a node is the collection of all finite subsets of the domain

of its child.

3. Classification, written $\triangle$ The domain of such a node is the disjoint union of the domains of its children.

In the next three subsections we show how to represent relational, network and hierarchical databases in the logical data model.

3.1.1. The Relational Model

Person	*Parent*
Rehoboam	Solomon
Solomon	David
Solomon	Batsheba
David	Jesse

Figure 1: The *Person–Parent* relation

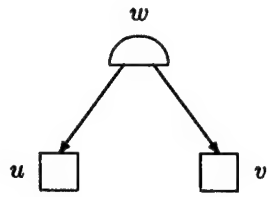


Figure 2: The *Person–Parent* relation as an LDM schema

Example 1: In most of the examples in this thesis the database will be a genealogy. Fig. 1 shows this database as a relation, together with the data in it.

The LDM schema that corresponds to it is shown in Fig. 2. It consists of two nodes u and v of type $\square$ that correspond to the *Person* and *Parent* attributes respectively, and one node w of type $\frown$ that contains pairs of related attributes.

For the moment, an instance **I** of an LDM schema will be an assignment to each node u of a set $I(u)$ of values from the corresponding domain (we shall modify the definition of an instance in Section 3.1.4). An instance of the LDM schema corresponding to the data in Fig. 1 consists of the following assignments:

$$\begin{aligned} I(u) &= \{\text{Rehoboam}, \text{Solomon}, \text{David}\} \\ I(v) &= \{\text{Solomon}, \text{David}, \text{Batsheba}, \text{Jesse}\} \end{aligned}$$

and

$$\begin{aligned} I(w) = \{&(\text{Rehoboam}, \text{Solomon}), (\text{Solomon}, \text{David}), (\text{Solomon}, \text{Batsheba}), \\ &(\text{David}, \text{Jesse})\} \end{aligned}$$

In general any relation R with attributes $A_1, \ldots, A_n$ can be converted into an LDM schema in a similar way. The corresponding schema will have one $\frown$-node for R, with n children of type $\square$, one corresponding to each attribute.

3.1.2. The Network Model

Example 2: The genealogy could be represented by the network in Fig. 3. In this network there are two record types, *Person* containing the names of the people in the database, and a dummy record *PP*. There are two links (sets) that connect each dummy record to a person and his parents.

The idea behind the mapping from the network to the LDM schema in Fig. 4 is as follows. Each record type R_i is mapped into a $\frown$-node v_{R_i}. For each field of R_i, v_{R_i} has a child of type $\square$. For each link (set) in the network with R_i as a member, let R_j be the owner of the link. Then v_{R_j} is a child of v_{R_i}.

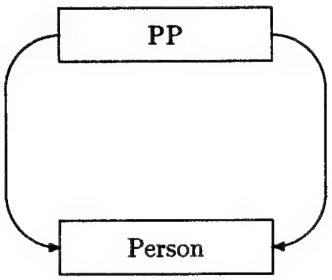


Figure 3: The genealogy as a network

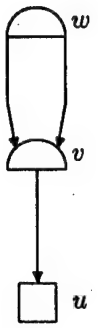


Figure 4: LDM schema corresponding to Fig. 3

In Fig. 4, w is v_{PP} and v is v_{Person}. u corresponds to the field of the *Person* record, i.e., the person's name, and the two arcs from w to v correspond to the two links.

If the network had the same contents as the relation in Fig. 1, the corresponding instance of the LDM schema in Fig. 4 would be

$$\begin{aligned} I(u) &= \{\text{Rehoboam, Solomon, David, Batsheba, Jesse}\} \\ I(v) &= \{(\text{Rehoboam}), (\text{Solomon}), (\text{David}), (\text{Batsheba}), (\text{Jesse})\} \end{aligned}$$

and

$$\begin{aligned} I(w) = \{&((\text{Rehoboam}), (\text{Solomon})), ((\text{Solomon}), (\text{David})), \\ &((\text{Solomon}), (\text{Batsheba})), ((\text{David}), (\text{Jesse}))\} \end{aligned}$$

3.1.3. The Hierarchical Model

Example 3: Fig. 5 shows a hierarchical representation of the genealogy. In this hierarchy, each *Person* record is related to the linked list of his parents. Even though the hierarchical model uses linked lists, this is really just a matter of the implementation, and intuitively the user should see only the connection between a person and the set of his parents. We therefore map each record type R_i into a ⌓-node v_{R_i} as we did for the network model, with a child of type □ corresponding to each of its fields. However, if R_i is a member of the link (R_i, R_j), then instead of connecting v_{R_i} to v_{R_j} directly, we connect them through a node of type ○.

Fig. 6 shows the LDM schema that we get from the hierarchy in Fig. 5. In this schema u_1 is v_{Person}, v_1 is v_{Parent}, u_2 and v_2 correspond to the fields of these records, and w is used to relate *Person* records to sets of *Parent* records.

The instance of Fig. 6 that corresponds to the data in the relation in Fig. 1 is

$$\begin{aligned} I(u_2) &= \{\text{Rehoboam, Solomon, David}\} \\ I(v_2) &= \{\text{Solomon, David, Batsheba, Jesse}\} \\ I(v_1) &= \{(\text{Solomon}), (\text{David}), (\text{Batsheba}), (\text{Jesse})\} \\ I(w) &= \{\{(\text{Solomon})\}, \{(\text{David}), (\text{Batsheba})\}, \{(\text{Jesse})\}\} \end{aligned}$$

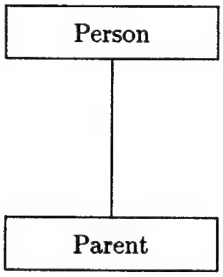


Figure 5: The genealogy as a hierarchy

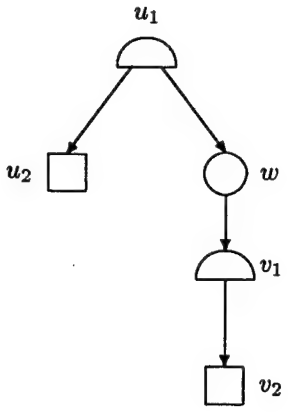


Figure 6: LDM schema corresponding to Fig. 5

and

$$I(x) = \{(\text{Rehoboam}, \{(\text{Solomon})\})(\text{Solomon}, \{(\text{David}), (\text{Batsheba})\}) \\ (\text{David}, \{(\text{Jesse})\})\}$$

Example 4: In practice we would probably not use the hierarchy of Fig. 5 as a representation of the genealogy, since it contains a lot of duplicated information. If a person appears in the database as both a child and as a parent, he will appear in both the *Person* and *Parent* records. For this reason, we would probably use a hierarchy with virtual records, as shown in Fig. 7. The corresponding LDM schema is then the cyclic schema in Fig. 8.

If the contents of the database are the same as before, the corresponding instance of the LDM schema is

$$\begin{aligned} I(u) &= \{\text{Rehoboam}, \text{Solomon}, \text{David}, \text{Batsheba}, \text{Jesse}\} \\ I(v) &= \{(\text{Jesse}, \emptyset), \\ &\quad (\text{David}, \{(\text{Jesse}, \emptyset)\}), \\ &\quad (\text{Batsheba}, \emptyset), \\ &\quad (\text{Solomon}, \{(\text{David}, \{(\text{Jesse}, \emptyset)\}), (\text{Batsheba}, \emptyset)\}), \\ &\quad (\text{Rehoboam}, \{(\text{Solomon}, \{(\text{David}, \{(\text{Jesse}, \emptyset)\}), (\text{Batsheba}, \emptyset)\})\})\} \\ I(w) &= \{\emptyset, \{(\text{Jesse}, \emptyset)\}), \\ &\quad \{(\text{David}, \{(\text{Jesse}, \emptyset)\}), (\text{Batsheba}, \emptyset)\}, \\ &\quad \{(\text{Solomon}, \{(\text{David}, \{(\text{Jesse}, \emptyset)\}), (\text{Batsheba}, \emptyset)\})\})\} \end{aligned}$$

3.1.4. Instances of LDM Schemas

As we see in Example 4, when the schema is cyclic and the nesting depth is large an instance can be rather complicated. If the data as well as the schema was cyclic, then the nesting depth would be infinite and we would not be able to write the instance down at all. This is similar to one of the problems with Jacobs'

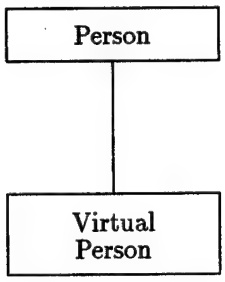


Figure 7: The genealogy as a hierarchy with virtual records

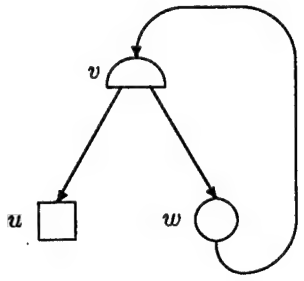


Figure 8: LDM schema corresponding to Fig. 7

database logic. The mathematical theory we develop to deal with this problem is closely related to the non well-founded sets of [Acz85]. Our approach to defining an instance of a schema is to model abstractly the concept of memory addresses and their contents. We use the term "l-values" for the abstract memory addresses, and the term "r-values" for their contents. An instance **I** then consists of two parts

1. An assignment of a set $I(u)$ of l-values (abstract addresses) to each node u of the schema.
2. An assignment of an r-value $r(l)$ to each l-value l in $I(u)$.

These l-values are taken from a fixed set L which will usually be the set of natural numbers. We now show what some of the instances in the previous examples look like when we use l-values and r-values.

Example 5: The instance of the schema in Example 1 consists of the following assignment of l-values to nodes.

$$\begin{aligned} I(u) &= \{1,2,3\} \\ I(v) &= \{4,5,6,7\} \end{aligned}$$

and

$$I(w) = \{8,9,10,11\}$$

We then assign an r-value $r(l)$ to each of these l-values. This assignment is shown in Fig. 9.

Example 6: In Fig. 10 we show the instance using l-values and r-values that corresponds to the instance of Example 4. Fig. 11 shows the links between the l-values and their r-values pictorially.

3.1.5. The Entity-Relationship Model

We conclude this section by showing how the logical data model can also be used to describe data structured by the Entity-Relationship Model of [Che76].

To map an entity-relationship schema into an LDM schema, we represent each entity type as a □-node, and each relationship record as a ⌓-node. A 1–1 arc from a relationship record to an entity type is represented by an edge from the corresponding ⌓-node to the corresponding □-node, while for a many to

$I(u)$

l	$r(l)$
1	Rehoboam
2	Solomon
3	David

$I(v)$

l	$r(l)$
4	Solomon
5	David
6	Batsheba
7	Jesse

$I(w)$

l	$r(l)$
8	$(1,4)$
9	$(2,5)$
10	$(2,6)$
11	$(3,7)$

Figure 9: Instance of the LDM schema that corresponds to a relation

$I(u)$

l	$r(l)$
1	Rehoboam
2	Solomon
3	David
4	Batsheba
5	Jesse

$I(v)$

l	$r(l)$
6	$(1,11)$
7	$(2,12)$
8	$(3,13)$
9	$(4,14)$
10	$(5,14)$

$I(w)$

l	$r(l)$
11	$\{7\}$
12	$\{8,9\}$
13	$\{10\}$
14	$\emptyset$

Figure 10: Instance of the LDM schema that corresponds to a hierarchy

one arc the connection is through a O-node. Figures 12 and 13 show two examples of entity-relationship database schemas from [Che76], together with the corresponding LDM schemas.

3.2. Query Languages

In this section, we give some examples of logical and algebraic queries on LDM schemas. All these examples are of queries that we can write in the query languages that we shall describe later on. The languages we describe later, however, are more formal, and therefore harder to use. The analogous situation in the relational model, is the comparison between Codd's tuple calculus and languages like QUEL. The languages in the current section have not been fully developed, and we describe them mainly as motivation for the formal presentation in the following chapters. In all the examples in this section, the database schema will be the LDM representation of the hierarchy, i.e., the schema in Fig. 14, together with the instance in Fig. 15.

3.2.1. The Logical Query Language

Both the logical and algebraic query languages have the property that the result can have a more general structure than a relation—in fact it is structured according to some LDM schema that is specified as part of the query. A query consists therefore of a specification of the nodes of the query, together with some QUEL-like statements specifying the contents of these nodes.

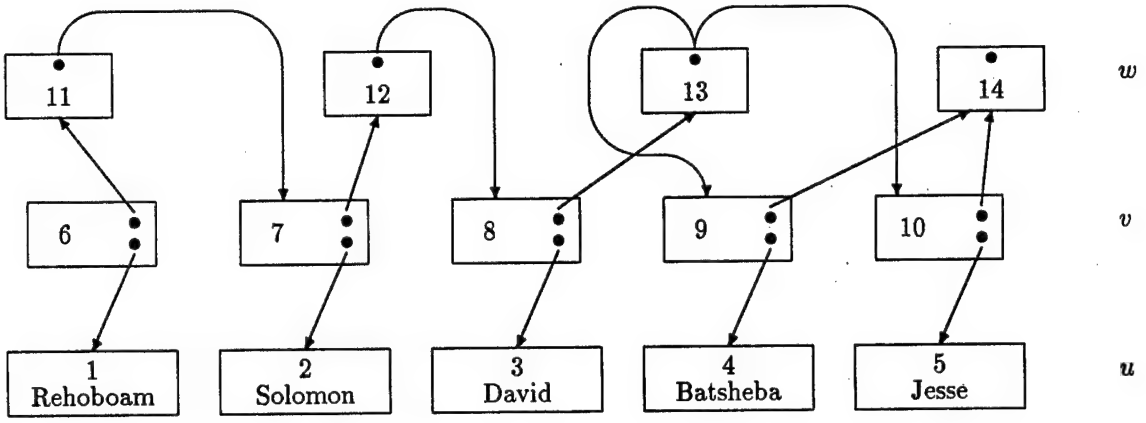


Figure 11: Pictorial representation of the instance in Fig. 10

Example 7: Our first query adds a new node Par-Sol of type O with child Person (see Fig. 16). This node contains the set of parents of "Solomon." The query is

```
type of Par-Sol is (collect,Person)
range of t is PP
range of u is PP
retrieve S into Par-Sol
where S={u.Person}
and t.Person=''Solomon''
and u is in t.Parents.
```

Example 8: In this example, we show how to restructure the database in the form shown in the left part of Fig. 17. We first copy all the people in the node Person into the node Pers

```
type of Pers is basic
range of t is PP
retrieve t.Person into Pers
```

The node Pars then contains all pairs that correspond to Person-Parent pairs.

```
type of Pars is (composition,Pers,Pers)
range of t is PP
range of u is PP
retrieve (t.Person,u.Person) into Pars
where u is in t.Parents.
```

3.2.2. The Algebraic Query Language

Example 9: We show how we could compute the query of Example 7 by a sequence of algebraic operations.

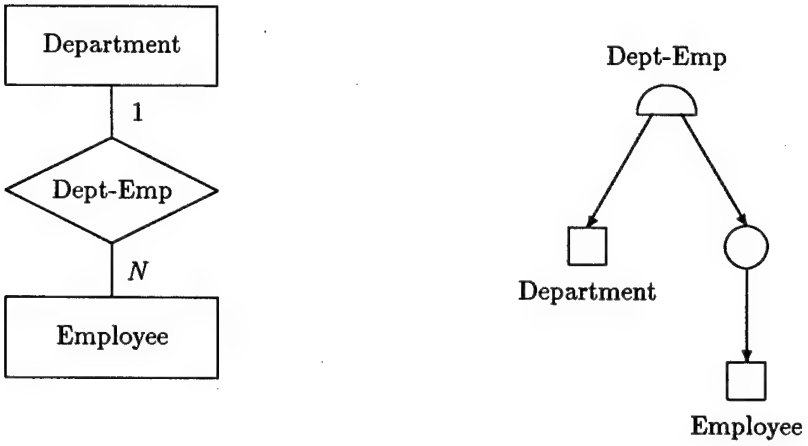


Figure 12: Department-Employee example

1. Select those elements of *PP* whose first component is "Solomon," i.e., $t_1 = \sigma_{(\text{Person}=\text{“Solomon”})}(\text{PP})$ (Fig. 18).
2. Do another type of selection: Select those sets that actually appear in tuples in t_1. This is the operation $t_2 = \sigma_{\text{Parents in}}(t_1)$ (Fig. 19).
3. We now have almost what we want, the only difference being that t_2 contains elements of *PP* rather than of *Person*. We have to do a dereferencing step, i.e., project onto *Person*. The operation is $t_3 = \Pi_{\text{Person}}(t_2)$ (Fig. 20).

The entire query is therefore

$$\Pi_{\text{Person}}\sigma_{\text{Parents in}}\sigma_{(\text{Person}=\text{“Solomon”})}(\text{PP})$$

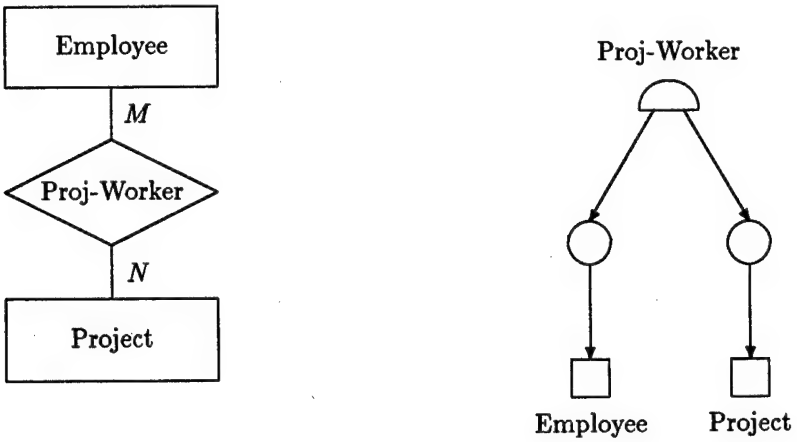


Figure 13: Project-Worker example

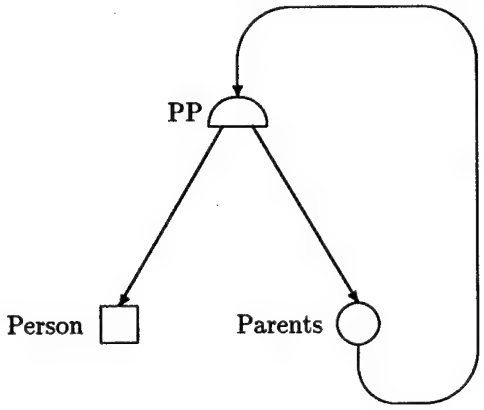


Figure 14: LDM Schema

I(Person)

l	$r(l)$
1	Rehoboam
2	Solomon
3	David
4	Batsheba
5	Jesse

I(PP)

l	$r(l)$
6	$(1, 11)$
7	$(2, 12)$
8	$(3, 13)$
9	$(4, 14)$
10	$(5, 14)$

I(Parents)

l	$r(l)$
11	$\{7\}$
12	$\{8, 9\}$
13	$\{10\}$
14	$\emptyset$

Figure 15: Instance of Fig. 14

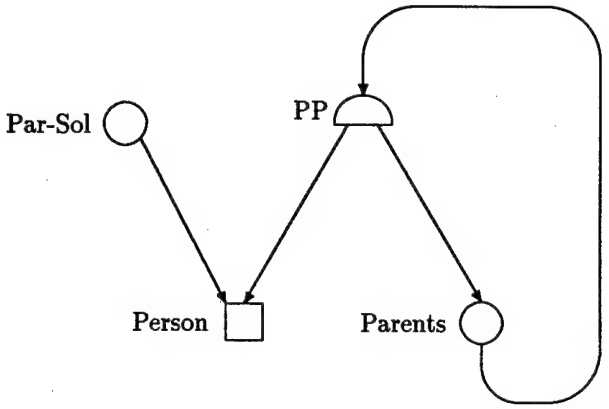


Figure 16: Example of a logical query

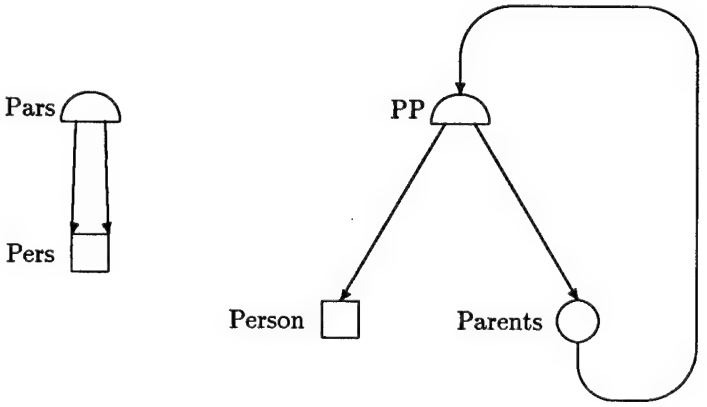


Figure 17: Another example of a logical query

$I(t_1)$

l	$r(l)$
15	(2,12)

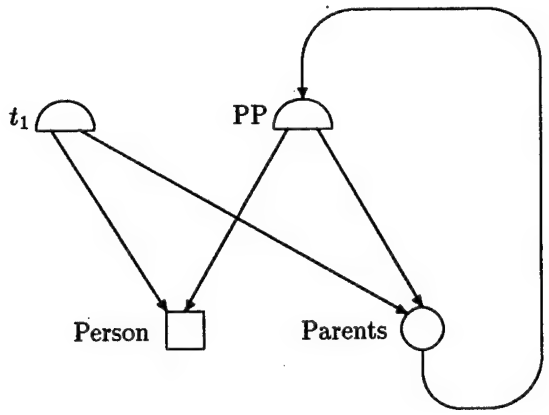


Figure 18: First step of the algebraic query

$I(t_2)$

l	$r(l)$
16	{8,9}

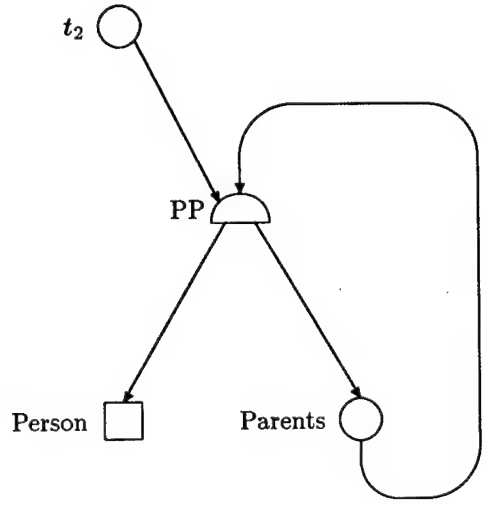


Figure 19: Second step of the algebraic query

$I(t_3)$

l	$r(l)$
17	{3,4}

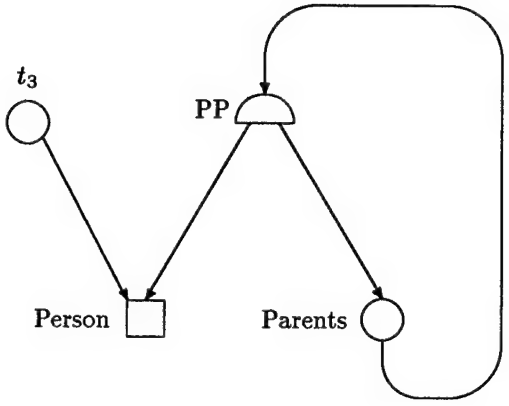


Figure 20: Third step of the algebraic query

Chapter 4

LDM Schemas and Instances

In this chapter we start the formal description of the logical data model. We define the two basic components of the model: LDM schemas, that describe how data is structured, and instances of these schemas.

4.1. LDM Schemas

The definition of a schema is essentially the same as outlined in the previous chapter. We have to go into several technical details that were not mentioned there. If v is a node of type $\bigcirc$, its domain consists of tuples formed from its children. For this to be meaningful we need an order on these children. Since there may be more than one edge between v and a node w, we also need an order on the occurrences of w in these tuples, so that what we really need is an order on all the *edges* with tail v. For simplicity, instead of using one order per node v we shall use a total order on all the edges of the schema.

Another technical detail is that a schema includes a set of constants. The reason for this is that we want to have a precise analogy between schemas and instances, on the one hand, and logical theories and models, on the other. The set of constants plays the role of individual constants in a logical theory.

Definition 1: A *schema* is a tuple $\mathbf{S} = \langle V, E, <, \mu, C \rangle$ where:

1. (V, E) is a directed multigraph.
2. $<$ is a total order on E.
3. μ is a function from the set of nodes V to the set of types $\{\square, \mathrm{O}, \bigcirc, \triangle\}$, that satisfies the following conditions (see Fig. 21)
   (a) $\mu(v) = \square$ iff v is a leaf.
   (b) If $\mu(v) = \mathrm{O}$, then v has exactly one child.
   (c) If $\mu(v) = \triangle$ then the children of v are distinct nodes (if $\mu(v) = \bigcirc$, however, there can be multiple edges from v to a node w).
4. C is a (possibly empty) set of *constants*.

$\mu(v)$ is called the *type* of v. For readability, we use the following abbreviations

1. $\mu(v) = (\mathrm{O}, w)$ is an abbreviation for "$\mu(v) = \mathrm{O}$ and its child is w."
2. (a) $\mu(v) = (\bigcirc, n)$ is an abbreviation for "$\mu(v) = \bigcirc$ and v has n children."

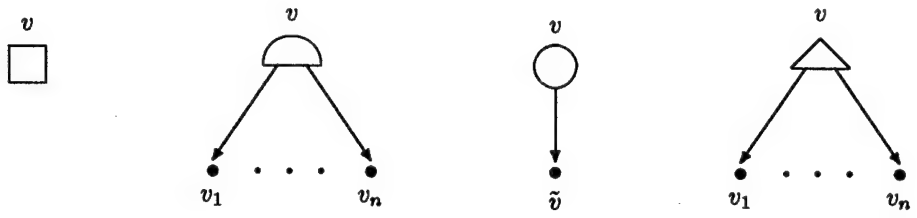


Figure 21: Nodes in LDM schemas

(b) $\mu(v) = (\frown, n, v_1, \ldots, v_n)$ is an abbreviation for "$\mu(v) = \frown$, there are exactly n edges $e_1, \ldots, e_n$ with tail v, these edges are in the order $e_1 < \cdots < e_n$ and their heads are $v_1, \ldots, v_n$."

3. (a) $\mu(v) = (\triangle\, n)$ is an abbreviation for "$\mu(v) = \triangle$ and v has n children."

   (b) $\mu(v) = (\triangle\, n, v_1, \ldots, v_n)$ is an abbreviation for "$\mu(v) = \triangle$, there are exactly n edges $e_1, \ldots, e_n$ with tail v, these edges are in the order $e_1 < \cdots < e_n$ and their heads are $v_1, \ldots, v_n$."

Some other abbreviations that we shall use include referring to elements of V and E as nodes and edges, respectively, of **S**, and referring to $<$ as an order on the children of a node of **S**. We shall ignore the order $<$ when it is clear from the context, and we shall often refer to a schema as $\langle V, E, \mu, C\rangle$.

As we outlined in the previous chapter, one part of a query on an LDM schema **S** is the addition of some nodes to **S**. We formalize this as follows

Definition 2: Let $\mathbf{S} = \langle V, E, <, \mu, C\rangle$ be a schema. $\mathbf{S}' = \langle V', E', <', \mu', C\rangle$ is an *extension* of **S** iff

1. $V \subseteq V'$
2. (a) $E \subseteq E'$

   (b) If $(v_1, v_2) \in E' - E$ then v_1 is in V', i.e. all new edges are either between new nodes, or from a new node to a node in V.

3. $<'|_{E\times E} = <$
4. $\mu' |_V = \mu$

4.2. Instances of LDM Schemas

Throughout this section $\mathbf{S} = \langle V, E, \mu, C\rangle$ will be a fixed LDM schema. An instance of **S** consists of two parts: An assignment of a set of objects called *l-values* to each node of **S**, and an assignment of an object called its *r-value* to each such l-value.

In the format model instances are constructed recursively from the leaves up. Since our model allows cycles, we cannot use this approach. What we do instead is define when a given object **I** is an instance.

Definition 3: An *instance* of **S** is a tuple $\mathbf{I} = \langle I, r, f\rangle$ that satisfies:

1. I is a function with domain V. This is the assignment of sets of l-values to nodes. We require that $I(v)$ and $I(w)$ be disjoint whenever v and w are distinct nodes of **S**.
2. r is a mapping with domain $\cup_{v\in V} I(v)$, i.e., from the set of all the l-values that are in the instance. The mapping r must satisfy:

(a) If $\mu(v) = (\ominus, n, v_1, \ldots, v_n)$ and $l \in I(v)$, then $r(l)$ is a tuple $(l_1, \ldots, l_n)$ such that for each i, $1 \leq i \leq n$, l_i is an element of $I(v_i)$.

(b) If $\mu(v) = (\bigcirc, w)$ and $l \in I(v)$, then $r(l)$ is a subset of $I(w)$.

(c) If $\mu(v) = (\triangle n, v_1, \ldots, v_n)$ and $l \in I(v)$, then $r(l) \in I(v_1) \cup \cdots \cup I(v_n)$.

Note that in general there is no constraint on the range of r on nodes of type $\Box$.

3. f is a function with domain C. For each $c \in C$, $f(c)$ is the *interpretation* of the constant c. In general there is no constraint on the range of f.

If l is in $\cup_{v \in V} I(v)$, we say that it is an *l-value* in **I**, and $r(l)$ is called its *r-value*. The set $\cup_{v \in V} r[I(v)]$ is called the set of r-values in **I**.

Definition 4: A *finite instance* of **S** is an instance $\mathbf{I} = \langle I, r, f \rangle$ of **S** such that for each node v of **S**, $I(v)$ is finite.

In practice, except for the reduction to first-order logic in Sections 5.2 and 5.3, we shall only be interested in a restricted class of instances, those that correspond to real databases. Such an instance is finite, and the instance $I(v)$ of each node v is a set of natural numbers. For a given database schema, there is also a fixed set D from which the data is taken. If v is of type $\Box$ and $l \in I(v)$, $r(l)$ must belong to the set D. Furthermore, each constant $c \in C$ must also belong to D, and we do not distinguish between c and $f(c)$. In short, after Section 5.3 we shall talk about schemas $\langle V, E, \mu \rangle$ and instances $\langle I, r \rangle$, where all the l-values are natural numbers, and all the data and constants are taken from a fixed set D.

Definition 5: Let **I** be an instance of the schema **S**, and let v be a node of **S** of type $(\ominus, n, v_1, \ldots, v_n)$. Let l be any l-value in $I(v)$. If $1 \leq i \leq n$, then $\Pi_i(l)$ will be the i^{th} component of $r(l)$. We shall also use the notation $\Pi_{v_i}(l)$ for this component, whenever this does not result in any ambiguity.

The following definition is related to when we can compare two l-values, i.e., if v and w are nodes of **S**, $l_1 \in I(v)$ and $l_2 \in I(w)$, is it possible for l_1 and l_2 to have the same r-value?

Definition 6: We say that two nodes v and w in a schema **S** are *similar* iff they are of the same type and have the same children, i.e., if one of the following holds:

1. $\mu(v) = \mu(w) = \Box$.
2. For some node u, $\mu(v) = \mu(w) = (\bigcirc, u)$.
3. For some n and nodes $u_1, \ldots, u_n$, $\mu(v) = \mu(w) = (\ominus, n, u_1, \ldots, u_n)$.
4. For some n and nodes $u_1, \ldots, u_n$, $\mu(v) = \mu(w) = (\triangle n, u_1, \ldots, u_n)$.

We would like to be able to show that whenever $r(l_1) = r(l_2)$ for some $l_1 \in I(v)$ and $l_2 \in I(w)$, then v and w must be similar. However, this may not be true for v or w of type $\Box$. For example if $\mu(v) = \Box$, since there is no constraint on the range of the function r on $I(v)$, the r-value of l_1 may just happen to have the form of a tuple or set of l-values. The logic will be defined in such a way that we shall only be able to compare r-values of similar nodes, so that this will not cause any problems.

Let **S**$'$ be an extension of **S**. We define an extension of **I** to an instance of **S**$'$ as follows.

Definition 7: Let **S**$'$ be an extension of **S**, and let $\mathbf{I} = \langle I, r, f \rangle$ be an instance of **S**. We say that an instance $\mathbf{I}' = \langle I', r', f \rangle$ of **S**$'$ is an extension of **I** to **S**$'$ iff

1. For all v in V, $I'(v) = I(v)$.

2. If v is a node of $\mathbf{S}$ and $l \in I(v)$, then $r'(l) = r(l)$.

The proof of the following lemma is straightforward.

Lemma 1: Let $\mathbf{S}'$ be an extension of $\mathbf{S}$, and let $\mathbf{I}'$ be an instance of $\mathbf{S}'$. Then there is a unique instance $\mathbf{I}$ of $\mathbf{S}$ such that $\mathbf{I}'$ is the extension of $\mathbf{I}$ to $\mathbf{S}'$. This instance is called the *restriction* of $\mathbf{I}'$ to $\mathbf{S}$. ∎

We conclude this chapter with a definition of isomorphism. Two instances will be isomorphic if they are essentially the same, i.e., if they differ only by renaming of l-values. As we shall want to show that the result of a query is well-defined up to isomorphism, we give a stronger definition of isomorphism. Let $\mathbf{I}$ be an instance of $\mathbf{S}$, let $\mathbf{S}'$ be an extension of $\mathbf{S}$ and let $\mathbf{I}_1$ and $\mathbf{I}_2$ be extensions of $\mathbf{I}$ to $\mathbf{S}'$. We shall say that $\mathbf{I}_1$ and $\mathbf{I}_2$ are isomorphic relative to $\mathbf{S}$, if there is an isomorphism between $\mathbf{I}_1$ and $\mathbf{I}_2$ that leaves the elements of $\mathbf{I}$ fixed. In the case of a query, this will mean that an isomorphism relative to the database leaves the contents of the database fixed.

Definition 8: Let $\mathbf{S}'$ be an extension of $\mathbf{S}$ and let $\mathbf{I} = \langle I, r, f\rangle$ be an instance of $\mathbf{S}$. Let $\mathbf{I}_1 = \langle I_1, r_1\rangle$ and $\mathbf{I}_2 = \langle I_2, r_2\rangle$ be two extensions of $\mathbf{I}$ to $\mathbf{S}'$. We say that $\mathbf{I}_1$ and $\mathbf{I}_2$ are *isomorphic relative to* $\mathbf{S}$ iff there is a mapping

$$g: \bigcup_{v \in \mathbf{S}} I_1(v) \xrightarrow[\text{onto}]{1\text{–}1} \bigcup_{v \in \mathbf{S}} I_2(v)$$

such that

1. For each node v of $\mathbf{S}'$, g maps $I_1(v)$ onto $I_2(v)$.
2. For each node v of $\mathbf{S}$, g is the identity on $I(v)$.
3. If v is a node of $\mathbf{S}'$ and $l \in I_1(v)$, then
   (a) If v is of type $\square$, then $r_2(g(l)) = r_1(l)$.
   (b) If v is of type $(\frown, n)$, then

$$r_2\Big(g(l)\Big) = \Big(g\Big(\Pi_1\Big(r_1(l)\Big)\Big), \ldots, g\Big(\Pi_n\Big(r_1(l)\Big)\Big)\Big)$$

   (c) If v is of type $\triangle$ then $r_2(g(l)) = g(r_1(l))$.
   (d) If v is of type $\bigcirc$, then $g[r_2(l)] = r_1[g(l)]$.

As a special case of this definition we get the definition of ordinary isomorphism.

Definition 9: Let $\mathbf{I}_1 = \langle I_1, r_1\rangle$ and $\mathbf{I}_2 = \langle I_2, r_2\rangle$ be instances of $\mathbf{S}$. We say that $\mathbf{I}_1$ and $\mathbf{I}_2$ are *isomorphic* iff they are isomorphic relative to the empty schema, i.e., the schema with $V = E = \mu = \emptyset$.

Chapter 5

The LDM Logic

5.1. Definition of the Logic

In this chapter we define the LDM logic. Our goal is to define a logic that is similar to the relational tuple calculus. We then use this logic as part of the logical query language. As the logic will resemble the relational tuple calculus, we can also use it to specify integrity constraints on LDM schemas, and to define views.

Throughout this chapter $\mathbf{S} = \langle V, E, \mu, C \rangle$ will be a fixed schema, and $\mathbf{I} = \langle I, r, f \rangle$ will be a fixed instance of **S**, unless mentioned otherwise. Each variable in the LDM logic has a fixed *sort*, where the sorts are the elements of V. The sorts restrict the possible values that the variable may have. For example, if x is a variable of sort v then x can take only values in $I(v)$. The analogue to this in the relational calculus is a tuple variable that ranges over a specific relation. We shall usually write a variable with its sort as a subscript, e.g., x_v. Two variables with different subscripts will denote distinct variables, so that x_u will be a different variable from x_v. Even though variables range over l-values, we shall often say "the l-value of x_v" instead of "the value of x_v," and "the r-value of x_v" when what we really mean is "the r-value of the value of x_v."

Definition 10: The *atomic formulas over* **S** are the following:

1. $x_v \ \pi_t \ y_w$, where w is a node of type $\frown$ and v is its t^{th} child.
2. $x_v \ \rho \ y_w$, where w is a node of type $\triangle$ and v is one of its children.
3. $x_v \in y_w$, where w is of type $(\bigcirc, v)$.
4. $x_v =_l y_v$.
5. $x_v =_r y_w$, where v and w are similar nodes.
6. $x_v =_r c$, where c is an element of C, and v is of type $\square$.

The atomic formula $x_v \ \pi_t \ y_w$ means that the l-value of x_v is the t^{th} component of the r-value of y_w. Note that we have to mention which component of w we are referring to, since there may be multiple edges from w to v. However, we shall also write $x_v \ \pi_v \ y_w$ when this is unambiguous. $x_v \ \rho \ y_w$ means that the r-value of y_w is x_v. Since there are is only one edge from w to v, we use ρ rather than ρ_t. $x_v \in y_w$ means that x_v is a member of the r-value of y_w.

There are several different kinds of equality. $x_v =_l y_v$ means that the l-values of x_v and y_v are equal. Since $I(v)$ and $I(w)$ are disjoint whenever $v \neq w$, the logic has no atomic formula of the form $x_v =_l y_w$ for $v \neq w$. $x_v =_r y_w$ means that the r-values of x_v and y_w are equal. We restrict this to similar nodes to

prevent us from comparing r-values of $\square$-nodes to tuples or sets of l-values, as we explained near the end of Chapter 4. Finally, the atomic formula $x_v =_r c$ means that the r-value of x_v is equal to the interpretation of the constant c.

By the way, the subscripted r's in the fifth and the sixth cases have slightly different meanings. The first one refers to the r-value of both sides, and the second just to the left side of the formula. We decided that the slight confusion this may cause was preferable to using a more cumbersome notation such as $_l{=}_l$, $_r{=}_r$ and $_r{=}_l$.

Definition 11: A *well-formed LDM formula over a schema* **S** is:

1. An atomic formula
2. $\phi_1 \vee \phi_2$, where ϕ_1 and ϕ_2 are well-formed formulas.
3. $\neg\phi_1$, where ϕ_1 is a well-formed formula.
4. $(\forall x_v)\phi_1$, where ϕ_1 is a well-formed formula.

The free variables of ϕ are defined in the same way as in first-order logic.

As usual, we use $\phi_1 \wedge \phi_2$ as an abbreviation for $\neg(\neg\phi_1 \vee \neg\phi_2)$, and $(\exists x_v)\phi$ as an abbreviation for $\neg(\forall x_v)\neg\phi$. We also use $\phi_1 \Rightarrow \phi_2$ and $\phi_1 \Leftrightarrow \phi_2$ with the standard meanings. Another useful abbreviation is the following.

Definition 12: "$x_v =_r (x^1_{v_1}, \ldots, x^n_{v_n})$" where v is a node of type $(\square, n, v_1, \ldots, v_n)$ will mean "$x^1_{v_1}\ \pi_1\ x_v \wedge \cdots \wedge x^n_{v_n}\ \pi_n\ x_v$."

We now define satisfaction of LDM formulas. Let $\phi(x^1_{v_1}, \ldots, x^n_{v_n})$ be an LDM formula whose free variables are $x^1_{v_1}, \ldots, x^n_{v_n}$. Let $l_1, \ldots, l_n$ be an assignment of l-values to the free variables in the formula, i.e., each l_i is a member of the corresponding $I(v_i)$. $\models_{\mathbf{I}} \phi(l_1, \ldots, l_n)$ will mean that ϕ is satisfied by $l_1, \ldots, l_n$ in the instance **I**. When **I** is clear from the context, we shall write $\models$ instead of $\models_{\mathbf{I}}$.

Definition 13: Let $\phi(x^1_{v_1}, \ldots, x^n_{v_n})$ be a formula with free variables $x^1_{v_1}, \ldots, x^n_{v_n}$, and let $l_i \in I(v_i)$ for all i, $1 \le i \le n$. Then $\models_{\mathbf{I}} \phi(l_1, \ldots, l_n)$ iff the following hold:

1. If ϕ is $x^i_v\ \pi_t\ y^j_w$, then $\models_{\mathbf{I}} (x^i_v\ \pi_t\ x^j_w)(l_1, \ldots, l_n)$ iff $l_i = \Pi_t(l_j)$.
2. If ϕ is $x^i_v\ \rho\ y^j_w$, then $\models_{\mathbf{I}} (x^i_v\ \rho\ x^j_w)(l_1, \ldots, l_n)$ iff $l_i = r(l_j)$.
3. If ϕ is $x^i_v \in x^j_w$, then $\models_{\mathbf{I}} (x^i_v \in x^j_w)(l_1, \ldots, l_n)$ iff $l_i \in r(l_j)$.
4. If ϕ is $x^i_v =_l x^j_v$, then $\models_{\mathbf{I}} (x^i_v =_l x^j_v)(l_1, \ldots, l_n)$ iff $l_i = l_j$.
5. If ϕ is $x^i_v =_r x^j_w$, then $\models_{\mathbf{I}} (x^i_v =_r x^j_w)(l_1, \ldots, l_n)$ iff $r(l_i) = r(l_j)$.
6. If ϕ is $x^i_v =_r c$, then $\models_{\mathbf{I}} (x^i_v =_r c)(l_1, \ldots, l_n)$ iff $r(l_i) = f(c)$.
7. $\models_{\mathbf{I}} (\phi_1 \vee \phi_2)$ iff $\models_{\mathbf{I}} \phi_1$ or $\models_{\mathbf{I}} \phi_2$.
8. $\models_{\mathbf{I}} \neg\phi$ iff $\models_{\mathbf{I}} \phi$ does not hold.
9. If ϕ is a formula with free variables $x^1_{v_1}, \ldots, x^n_{v_n}, y_w$, then

$$\models_{\mathbf{I}} \Big((\forall y_w)\phi\Big)(l_1, \ldots, l_n) \text{ iff for all } l \in I(w), \quad \models_{\mathbf{I}} \phi(l_1, \ldots, l_n, l)$$

Definition 14: An LDM *constraint* or *sentence* is an LDM formula with no free variables.

Definition 15: A *constrained* schema is a pair $(\mathbf{S}, \phi)$, where $\mathbf{S}$ is a schema and ϕ is an LDM constraint over $\mathbf{S}$. An instance of $(\mathbf{S}, \phi)$ is an instance $\mathbf{I}$ of $\mathbf{S}$ that satisfies $\models_{\mathbf{I}} \phi$.

Definition 16: Let ϕ be an LDM sentence. We say that an instance $\mathbf{I}$ of $\mathbf{S}$ satisfies ϕ iff $\models_{\mathbf{I}} \phi$ holds.

Definition 17: Let Σ be a set of LDM sentences, and let ϕ be an LDM sentence. We say that $\Sigma \models \phi$ iff for every instance $\mathbf{I}$ of $\mathbf{S}$ that satisfies all the sentences in Σ, $\models_{\mathbf{I}} \phi$ holds.

Definition 18: Let ϕ be an LDM sentence. and let ϕ be an LDM sentence. We say that ϕ is *valid* iff for any instance $\mathbf{I}$ of $\mathbf{S}$, $\models_{\mathbf{I}} \phi$ holds.

Example 10: This example and the next one will be over the LDM schema of Fig. 8 (page 10) with the instance of Fig. 10 (page 11). The LDM formula $\phi(x_u, y_v) = (x_u \ \pi_1 \ y_v)$ says that the l-value of x_u is equal to the first component of the r-value of y_v. $\models_{\mathbf{I}} \phi(l_1, l_2)$ holds for the (l_1, l_2) pairs $(1,7)$, $(2,8)$, $(3,9)$, $(4,10)$, and $(5,11)$.

Example 11: Let us see how to write a constraint that says that each l-value of u is related to exactly one set in w. So for example, '8' and '9' as parents of '2' must be in one set rather that in two different sets. The constraint is

$$\phi = (\forall x_u)(\forall y_v^1)(\forall y_v^2)(\forall z_w^1)(\forall z_w^2)\Big(y_v^1 =_r (x_u, z_w^1) \wedge y_v^2 =_r (x_u, z_w^2) \Rightarrow z_w^1 =_l z_w^2\Big)$$

In other words, each l-value in u (x_u) has at most one l-value in w (z_w^1 and z_w^2) associated with it. This association is through y_v^1 and y_v^2.

Note that this constraint says that each *l-value* in u is associated with at most one set in w, rather than saying that each *person* in the database is associated with at most one such set. There could still be duplication in u, e.g., two l-values with the r-value "Solomon." One way to prevent this would be through the constraint

$$\psi = (\forall x_u^1)(\forall x_u^2)(x_u^1 =_r x_u^2 \Rightarrow x_u^1 =_l x_u^2)$$

The following lemma shows that we can restrict the logic without reducing its power. We show that there is no need for atomic formulas that compare r-values of internal nodes. This lemma will make some subsequent proofs and definitions much simpler.

Lemma 2: Let $\phi(x_{v_1}^1, \ldots, x_{v_n}^n)$ be an LDM formula whose free variables are the variables $x_{v_1}^1, \ldots, x_{v_n}^n$. There is an LDM formula $\psi(x_{v_1}^1, \ldots, x_{v_n}^n)$ with the same free variables, that does not contain any atomic subformula of the form $x_u =_r y_v$ with $\mu(v), \mu(w) \neq \square$. This formula is equivalent to ϕ, i.e., for all instances $\mathbf{I}$ of $\mathbf{S}$ and all $l_1, \ldots, l_n$, $l_i \in I(v_i)$, $\models_{\mathbf{I}} \phi(l_1, \ldots, l_n)$ iff $\models_{\mathbf{I}} \psi(l_1, \ldots, l_n)$.

Proof: The proof is by induction on the size of ϕ. We show how to construct ψ for formulas of the form $x_u =_r y_v$, where u and v are similar and not of type $\square$. The result will then follow immediately.

We distinguish between the possible types of v and w.

1. If u and v are of type $(\bigcirc, w)$, then $\psi(x_u, y_v)$ will be $(\forall z_w)(z_w \in x_u \Leftrightarrow z_w \in y_v)$, where z_w is some new variable. Let $\mathbf{I}$ be an instance of $\mathbf{S}$. Then

$$\begin{aligned} \models_{\mathbf{I}} (x_u =_r y_v)(l_1, l_2) &\Leftrightarrow r(l_1) = r(l_2) \\ &\Leftrightarrow \text{For all } l \text{ in } I(w), l \in r(l_1) \Leftrightarrow l \in r(l_2) \\ &\Leftrightarrow \models_{\mathbf{I}} \Big((\forall z_w)(z_w \in x_u \Leftrightarrow z_w \in y_v)\Big)(l_1, l_2) \end{aligned}$$

and therefore $\phi \Leftrightarrow \psi$ is valid.

2. If u and v are of type $(\Box, n, w_1, \ldots, w_n)$, then $\psi(x_u, y_v)$ will be

$$(\forall z^1_{w_1}) \cdots (\forall z^n_{w_n}) \Big((z^1_{w_1}\ \pi_1\ x_u \Leftrightarrow z^1_{w_1}\ \pi_1\ y_v) \wedge \cdots \wedge (z^n_{w_n}\ \pi_n\ x_u \Leftrightarrow z^n_{w_n}\ \pi_n\ y_v) \Big)$$

where $z^1_{w_1}, \ldots, z^n_{w_n}$ are n different new variables. Let $\mathbf{I}$ be an instance of $\mathbf{S}$. Then $\models_{\mathbf{I}} (x_u =_r y_v)(l_1, l_2)$ is equivalent to $r(l_1) = r(l_2)$. If $r(l_1) = (l^1_1, \ldots, l^n_1)$ and $r(l_2) = (l^1_2, \ldots, l^n_2)$, $r(l_1) = r(l_2)$ is equivalent to $l^i_1 = l^i_2$, for $i = 1, \ldots, n$. In other words, for each such i,

$$\models_{\mathbf{I}} \Big((\forall z^i_w)(z^i_{w_i}\ \pi_i\ x_u \Leftrightarrow z^i_{w_i}\ \pi_i\ y_v) \Big)(l_1, l_2)$$

and therefore $\models_{\mathbf{I}} (x_u =_r y_v)(l_1, l_2)$ is equivalent to

$$\models_{\mathbf{I}} (\forall z^1_{w_1}) \cdots (\forall z^n_{w_n}) \Big((z^1_{w_1}\ \pi_1\ x_u \Leftrightarrow z^1_{w_1}\ \pi_1\ y_v) \wedge \cdots \wedge (z^n_{w_n}\ \pi_n\ x_u \Leftrightarrow z^n_{w_n}\ \pi_n\ y_v) \Big)(l_1, l_2)$$

i.e., $\phi \Leftrightarrow \psi$ is valid.

3. If u and v are of type $(\triangle, n, w_1, \ldots, w_n)$, then $\psi(x_u, y_v)$ will be

$$(\exists z^1_{w_1})(z^1_{w_1}\ \rho\ x_u \wedge z^1_{w_1}\ \rho\ y_v) \vee \cdots \vee (\exists z^n_{w_n})(z^n_{w_n}\ \rho\ x_u \wedge z^n_{w_n}\ \rho\ y_v)$$

where $z^1_{w_1}, \ldots, z^n_{w_n}$ are n different new variables. Let $\mathbf{I}$ be an instance of $\mathbf{S}$. Then $\models_{\mathbf{I}} (x_u =_r y_v)(l_1, l_2)$ is equivalent to $r(l_1) = r(l_2) = l$. This can hold only if for some i, $1 \leq i \leq n$, $l \in I(w_i)$ in which case

$$\models_{\mathbf{I}} \Big((\exists z^i_{w_i})(z^i_{u_i}\ \rho\ x_u \wedge z^i_{w_i}\ \rho\ y_v) \Big)(l_1, l_2)$$

and therefore $\phi \Leftrightarrow \psi$ is again valid. ∎

From now on, we shall assume that $x_u =_r y_v$ can appear as a subformula only when $\mu(v) = \mu(w) = \Box$, as far as proofs and definitions are concerned. We shall continue to use the more general form when convenient.

The proof of the following lemma, that says that satisfaction is preserved under isomorphism, is straightforward.

Lemma 3: Let $\mathbf{S}'$ be an extension of $\mathbf{S}$, and let $\mathbf{I}_1$ and $\mathbf{I}_2$ be extensions of $\mathbf{I}$ to $\mathbf{S}'$. Let g be an isomorphism from $\mathbf{I}_1$ to $\mathbf{I}_2$ relative to $\mathbf{S}$, and let $\phi(x^1_{v_1}, \ldots, x^n_{v_n})$ be an LDM formula. Then

$$\models_{\mathbf{I}_1} \phi(l_1, \ldots, l_n) \Leftrightarrow \models_{\mathbf{I}_2} \phi\Big(g(l_1), \ldots, g(l_n)\Big)$$ ∎

Lemma 4: Let $\phi(x^1_{v_1}, \ldots, x^n_{v_n})$ be an LDM formula over $\mathbf{S}$ whose free variables are $x^1_{v_1}, \ldots, x^n_{v_n}$. Let $\mathbf{I}$ be a finite instance and let $l_i \in I(v_i)$ for all i, $1 \leq i \leq n$. Then $\models_{\mathbf{I}} \phi(l_1, \ldots, l_n)$ can be determined effectively.

Proof: We show this by induction on the size of the formula. For atomic formulas testing for satisfaction is straightforward. Testing for disjunction and negation is also clearly effective. For quantification we make use of the finiteness of $\mathbf{I}$. In order to test whether $\models_{\mathbf{I}} ((\forall y_w)\phi)(l_1, \ldots, l_n)$, we test whether $\models_{\mathbf{I}} \phi(l_1, \ldots, l_n, l)$ for each l in the finite set $I(w)$. ∎

5.2. The Relation between LDM logic and First-Order Logic

In this section we shall show that the LDM logic is essentially first-order; that is, it is compact and it satisfies a Löwenheim-Skolem theorem. We shall prove this by reducing LDM logic to a certain many-sorted first-order logical theory with equality. We mention in contrast that Jacobs' database logic [Jac82] is inherently a higher-order logic that does not have any of these properties. In the next section, we shall use this reduction to develop a proof theory for the LDM-logic. In both these sections we shall not assume that instances are finite, or make any of the other assumptions on instances that we mentioned earlier.

Let $\mathbf{L}$ be an LDM logic over $\mathbf{S}$. We construct a many-sorted first order logic with equality $\mathbf{L}'$ as follows. The sorts of $\mathbf{L}'$ are $V \cup \{\overline{c}\}$, i.e., we have a sort v for each node of the schema, and one special sort $\overline{c}$ that corresponds to the domain from which the data is taken. $\mathbf{L}'$ has variables ranging over all the sorts, except for the special sort, $\overline{c}$ since we do not want to be able to quantify over the data domain.

The relation symbols of $\mathbf{L}'$ are

$$\{\in_w \mid w \in V \text{ and } \mu(w) = \bigcirc\} \cup \{\rho_{w,v} \mid w \in V, \mu(w) = \triangle \text{ and } v \text{ is a child of } w\}$$

If $w \in V$ is of type $(\bigcirc, v)$ then $\in_w$ is a binary relation symbol between elements of sorts v and w. $\rho_{w,v}$ is also a binary relation between these sorts. We shall use infix notation for binary relations.

The function symbols of $\mathbf{L}'$ are

$$\{\pi_{w,i} \mid w \in V, \mu(w) = (\backsim, n), 1 \leq i \leq n\} \cup \{f_w \mid w \in V, \mu(w) = \square\}$$

The function symbol $\pi_{w,i}$ is from sort w to sort v, where v is the i^{th} child of w. We shall also use the notation $\pi_{w,v}$ when its meaning is unambiguous. The function symbol f_w is from sort w to sort $\overline{c}$. Intuitively, $\pi_{w,i}$ maps its argument to its i^{th} component, and f_w maps its argument to its r-value, which is a data element. The reason we use $\rho_{w,v}$, rather than a function symbol, is that ρ should be interpretated as a function from $I(w)$ to the union of the instances of its children, whereas in first-order logic all functions are to exactly one sort. For this reason we use a relation symbol for ρ, and we shall also need some extra axioms for $\mathbf{L}'$ besides the usual logical axioms. Finally, the constants of $\mathbf{L}$ (i.e., the elements of C) are also constants of $\mathbf{L}'$, of sort $\overline{c}$.

The logical theory $\mathbf{L}'$ then consists of the standard logical axioms, together with the set $\text{Ax}(\mathbf{S})$ of axioms for ρ. $\text{Ax}(\mathbf{S})$ contains the following axioms for each node w of $\mathbf{S}$ that is of type $(\triangle, n, v_1, \ldots, v_n)$.

1. $(\forall x_w)\Big((\exists y^1_{v_1})(y^1_{v_1}\ \rho_{w,v_1}\ x_w) \vee \cdots \vee (\exists y^n_{v_n})(y^n_{v_n}\ \rho_{w,v_n}\ x_w)\Big)$
2. For all i and j where $1 \leq i, j \leq n$ and $i \neq j$, $(\forall x_w)\Big((\exists y^i_{v_i})(y^i_{v_i}\ \rho_{w,v_i}\ x_w) \Rightarrow (\forall y^j_{v_j})\neg(y^j_{v_j}\ \rho_{w,v_j} x_w)\Big)$
3. For all i, $1 \leq i \leq n$, $(\forall x_w)(\forall y^1_{v_i})(\forall y^2_{v_i})\Big((y^1_{v_i}\ \rho_{w,v_i}\ x_w) \wedge (y^2_{v_i}\ \rho_{w,v_i}\ x_w) \Rightarrow (y^1_{v_i} =_l y^2_{v_i})\Big)$

Essentially these axioms say that the interpretation of ρ is a function from $I(w)$ to $I(v_1) \cup \cdots \cup I(v_n)$. When we use the symbol $\models$ in the theory $\mathbf{L}'$, e.g., $\Sigma \models \phi$, we shall mean that every model of Σ and $\text{Ax}(\mathbf{S})$ is also a model of ϕ.

We now define two mappings. The first, F (for "First-order") will map formulas and instances of the LDM logic $\mathbf{L}$ to formulas and structures of $\mathbf{L}'$. The second mapping, L (for "LDM") will map $\mathbf{L}'$ to $\mathbf{L}$.

5.2.1. Mapping LDM Logic into First-Order Logic

We first show how to map LDM formulas into first-order formulas.

Definition 19: Let ϕ be a formula of **L**. W.l.o.g., assume that it is in the form of Lemma 2 (page 24). $F(\phi)$ is the **L**′-formula defined as follows.

1. $F(x_v\ \pi_t\ y_w)$ is $x_v = \pi_{w,t}(y_w)$.
2. $F(x_v\ \rho\ y_w)$ is $x_v\ \rho_{w,v}\ y_w$.
3. $F(x_v \in y_w)$ is $x_v \in_w y_w$.
4. $F(x_v =_l y_v)$ is $x_v = y_v$.
5. $F(x_v =_r y_w)$ is $f_v(x_v) = f_w(y_w)$, where v and w are both of type $\square$.
6. $F(x_v =_r c)$ is $f_v(x_v) = c$.
7. $F(\phi_1 \wedge \phi_2) = F(\phi_1) \wedge F(\phi_2)$.
8. $F(\neg\phi) = \neg F(\phi)$.
9. $F((\forall x_v)\phi) = (\forall x_v)F(\phi)$.

We now map an instance **I** of **S** into a structure $F(\mathbf{I})$ over **L**′. An **L**′-structure **M** consists of an assignment of a domain $D_{\mathbf{M}}(s)$ to each sort s, an assignment of a function $g_{\mathbf{M}}$ to each function symbol g, an assignment of a relation $R_{\mathbf{M}}$ to each relation symbol R and an interpretation of each individual constant of **L**′.

Definition 20: Let $\mathbf{I} = \langle I, r, f\rangle$ be an instance of **S**. $F(\mathbf{I})$ is the following **L**′-structure.

1. The domain corresponding to each sort v of **L**′, except for the sort $\overline{c}$, is the set $I(v)$. Formally, $D_{F(\mathbf{I})}(v) = I(v)$.
2. $D_{F(\mathbf{I})}(\overline{c}) = \{f(c) \mid c \in C\} \cup \{r(l) \mid l \in I(v) \text{ and } \mu(l) = \square\}$. This means that the domain that corresponds to the sort $\overline{c}$ consists of the interpretation of all the logical constants and of all the data in the instance.
3. The interpretation of $\pi_{w,i}$ is the function $(\pi_{w,i})_{F(\mathbf{I})}$ that maps each element of $I(w)$ to the i^{th} component of its r-value. Formally, $(\pi_{w,i})_{F(\mathbf{I})}(l) = \Pi_i(l)$, for all $l \in I(w)$.
4. The interpretation of $\rho_{w,v}$ is the relation

$$(\rho_{w,v})_{F(\mathbf{I})} = \Big\{(l_1, l_2) \mid l_1 \in I(v), l_2 \in I(w) \text{ and } l_1 = r(l_2)\Big\}$$

5. The interpretation of f_v is the function $(f_v)_{F(\mathbf{I})}$ that maps each element of $I(v)$ to its r-value, i.e., $(f_v)_{F(\mathbf{I})}(l) = r(l)$ for all $l \in I(v)$. Note that all these r-values are in $D_{F(\mathbf{I})}(\overline{c})$ by 2.
6. The interpretation of $\in_w$ where w is of type $(\bigcirc, v)$ is the relation

$$(\in_w)_{F(\mathbf{I})} = \Big\{(l_1, l_2) \mid l_1 \in I(v), l_2 \in I(w) \text{ and } l_1 \in r(l_2)\Big\}$$

7. The interpretation of the individual constant c is $f(c)$. This is in $D_{F(\mathbf{I})}(\overline{c})$ by 2.

This definition immediately implies the following lemma.

Lemma 5: $\models_{F(\mathbf{I})} \mathrm{Ax}(\mathbf{S})$ ∎

Theorem 6: For any **L**-sentence ϕ, $\models_{\mathbf{I}} \phi \Leftrightarrow \models_{F(\mathbf{I})} F(\phi)$.

Proof: The proof is by induction on the size of ψ. The induction hypothesis is as follows. If $\psi(x^1_{v_1}, \ldots, x^n_{v_n})$ is an L-formula and $l_i \in I(v_i)$ for all i, $1 \leq i \leq n$, then

$$\models_{\mathbf{I}} \psi(l_1, \ldots, l_n) \Leftrightarrow \models_{F(\mathbf{I})} F(\psi)(l_1, \ldots, l_n)$$

The theorem follows immediately by taking $\psi = \phi$.

For atomic formulas the proof of the induction hypothesis is straightforward. For example

$$\begin{aligned} \models_{\mathbf{I}} (x_v \ \pi_t \ y_w)(l_v, l_w) &\Leftrightarrow l_v = \Pi_t(l_w) \\ &\Leftrightarrow l_v = (\pi_{w,t})_{F(\mathbf{I})}(l_w) \\ &\Leftrightarrow \models_{F(\mathbf{I})} (x_v = \Pi_{w,t}(y_w))(l_v, l_w) \\ &\Leftrightarrow \models_{F(\mathbf{I})} F(x_v \ \pi_t \ y_w)(l_v, l_w) \end{aligned}$$

If ψ is either $\psi_1 \vee \psi_2$ or $\neg\psi_1$, the proof is easy. Finally, if $\psi(x^1_{v_1}, \ldots, x^n_{v_n})$ is the formula $(\forall y_w)\chi(x^1_{v_1}, \ldots, x^n_{v_n}, y_w)$, then

$$\models_{\mathbf{I}} \Big((\forall y_w)\chi\Big)(l_1, \ldots, l_n) \Leftrightarrow \text{For all } l \text{ in } I(w), \quad \models_{\mathbf{I}} \chi(l_1, \ldots, l_n, l)$$

By the induction hypothesis and the definition of $D_{F(\mathbf{I})}(w)$,

$$\begin{aligned} &\Leftrightarrow \text{For all } l \text{ in } D_{F(\mathbf{I})}(w), \quad \models_{F(\mathbf{I})} F(\chi)(l_1, \ldots, l_n, l) \\ &\Leftrightarrow \models_{F(\mathbf{I})} F(\psi)(l_1, \ldots, l_n) \ \blacksquare \end{aligned}$$

5.2.2. Mapping the First-Order Logic into LDM Logic

In order to define the inverse mapping L from $\mathbf{L}'$ to $\mathbf{L}$, we first examine the form of atomic formulas in the first-order logic $\mathbf{L}'$. Since the only relation symbols in $\mathbf{L}'$ are $\in_w$, $\rho_{w,v}$ and $=$, such an atomic formula must be one of the following.

1. $t_1 \in_w t_2$ where w is of type $(\bigcirc, v)$, t_1 is of sort v and t_2 is of sort w.
2. $t_1 \ \rho_{w,v} \ t_2$ where w is of type $\triangle$, v is a child of w, t_1 is of sort v and t_2 is of sort w.
3. $t_1 = t_2$ where both t_1 and t_2 are of sort w for some w in V.
4. $t_1 = t_2$ where both t_1 and t_2 are of sort $\overline{c}$.

Note that we cannot have $t_1 \in_w t_2$ or $t_1 \ \rho_{w,v} \ t_2$ where either t_1 or t_2 is of sort $\overline{c}$.

We first introduce some notation. Whenever t is a term of the form

$$\pi_{u_2,u_1} \cdots \pi_{u_{n-1},u_{n-2}} \pi_{u_n,u_{n-1}}(x_{u_n})$$

we shall want to replace it by a variable of sort u_1. For this purpose, we introduce new variables $z^1_{u_1}$, ..., $z^{n-1}_{u_{n-1}}$ of sort u_1, u_2, ..., u_{n-1} respectively. Q_t will stand for the sequence of quantifiers

$$Q_t = (\exists z^1_{u_1}) \cdots (\exists z^{n-1}_{u_{n-1}})$$

and ψ_t will say that $z^1_{u_1}$, ..., $z^{n-1}_{u_{n-1}}$ are on the path from x_{u_n}, i.e.,

$$\Big((z^1_{u_1} \pi_{u_2,u_1} z^2_{u_2}) \wedge \cdots \wedge (z^{n-1}_{u_{n-1}} \pi_{u_n,u_{n-1}} x_{u_n})\Big)$$

Using this notation we define $L(\phi)$ for an atomic $\mathbf{L}'$-formula ϕ as follows.

1. Since the result of each f_u is of sort $\bar{c}$ and there are no function symbols from sort $\bar{c}$, whenever ϕ is $t_1 \in_w t_2$ or $t_1 \rho_{w,v} t_2$, the only possible form that the terms t_1 and t_2 can have is

$$t_1 = \pi_{u_1,v}\pi_{u_2,u_1}\cdots\pi_{u_n,u_{n-1}}\pi_{u,u_n}(x_u)$$

and

$$t_2 = \pi_{u'_1,w}\pi_{u'_2,u'_1}\cdots\pi_{u'_m,u'_{m-1}}\pi_{u',u'_m}(y_{u'})$$

(n or m may be equal to 0, in which case some of the new variables are not needed. It should be obvious how to modify the definitions in this case, and in the case when t_1 or t_2 is an individual constant.) We now define

$$L(t_1 \in_w t_2) = Q_{t_1}Q_{t_2}\Big(\psi_{t_1} \wedge \psi_{t_2} \wedge (z_v \in z_w)\Big)$$

where z_v and z_w are the new variables of sorts v and w that we introduced.

In a similar way $L(t_1 \rho_{w,v} t_2)$ is defined as as

$$Q_{t_1}Q_{t_2}\Big(\psi_{t_1} \wedge \psi_{t_2} \wedge (z_v \rho z_w)\Big)$$

2. When ϕ is $t_1 = t_2$ where t_1 and t_2 are of sort w, t_1 and t_2 must be of the form

$$t_1 = \pi_{u_1,w}\pi_{u_2,u_1}\cdots\pi_{u_n,u_{n-1}}\pi_{u,u_n}(x_u)$$

and

$$t_2 = \pi_{v_1,w}\pi_{v_2,v_1}\cdots\pi_{v_m,v_{m-1}}\pi_{v,v_m}(y_v)$$

We then define

$$L(t_1 = t_2) = Q_{t_1}Q_{t_2}\Big(\psi_{t_1} \wedge \psi_{t_2} \wedge (z^1_w =_l z^2_w)\Big)$$

where z^1_w and z^2_w are the two new variables of sort w that we introduced.

3. When ϕ is $t_1 = t_2$ where t_1 and t_2 are of sort $\bar{c}$, t_1 and t_2 must have the form

$$t_1 = f_{u_1}\pi_{u_2,u_1}\cdots\pi_{u_n,u_{n-1}}\pi_{u,u_n}(x_u)$$

and

$$t_2 = f_{v_1}\pi_{v_2,v_1}\cdots\pi_{v_m,v_{m-1}}\pi_{v,v_m}(y_v)$$

Write $t_1 = f_{u_1}(t_3)$ and $t_2 = f_{v_1}(t_4)$. We then define $L(t_1 = t_2)$ to be

$$Q_{t_3}Q_{t_4}\Big(\psi_{t_3} \wedge \psi_{t_4} \wedge (z^1_{u_1} =_r z^2_{v_1})\Big)$$

Definition 21: When ϕ is an $\mathbf{L}'$-formula, $L(\phi)$ is defined as follows.

1. If ϕ is an atomic formula $L(\phi)$ is defined above.
2. $L(\phi_1 \wedge \phi_2) = L(\phi_1) \wedge L(\phi_2)$.
3. $L(\neg\phi) = \neg L(\phi)$.
4. $L((\forall x_v)\phi) = (\forall x_v)L(\phi)$. The fact that $\mathbf{L}'$ has no variables of sort $\bar{c}$ is necessary to guarantee that this is an $\mathbf{L}$-formula.

We now show how to map $\mathbf{L}'$-structures into $\mathbf{L}$-instances.

Definition 22: Let $\mathbf{M}$ be an $\mathbf{L}'$-structure that satisfies Ax(S). We define $L(\mathbf{M}) = \left\langle I_{L(\mathbf{M})}, r_{L(\mathbf{M})}, f_{L(\mathbf{M})} \right\rangle$ to be the following instance of $\mathbf{S}$.

1. For each node v of $\mathbf{S}$, $I_{L(\mathbf{M})}(v)$ is the domain that corresponds to the sort v, i.e., $D_{\mathbf{M}}(v)$.
2. For each $l \in I(v)$, $r(l)$ is defined as follows:
   (a) If $\mu(v) = \square$, then $r(l) = (f_v)_{\mathbf{M}}(l)$.
   (b) If $\mu(v) = (\backsim, n, v_1, \ldots, v_n)$, then $r(l) = \Big((\pi_{v,v_1})_{\mathbf{M}}(l), \ldots, (\pi_{v,v_n})_{\mathbf{M}}(l)\Big)$.
   (c) If $\mu(v) = (\bigcirc, w)$, then $r(l) = \{\tilde{l} \mid \tilde{l}\ (\in_v)_{\mathbf{M}} l\}$.
   (d) If $\mu(v) = (\triangle n, v_1, \ldots, v_n)$, then $r(l) = \tilde{l}$ where $\tilde{l}$ is the unique element of $I(v_1) \cup \cdots \cup I(v_n)$ such that $\tilde{l}\ \rho_{w,v_i}\ l$ for some i. The existence and uniqueness of $\tilde{l}$ are consequences of Ax(S).
3. For each $c \in C$, $f(c)$ is the interpretation of the individual constant c in the structure $\mathbf{M}$.

Lemma 7: $L(\mathbf{M})$ is well defined and is an instance of $\mathbf{S}$. ∎

Theorem 8: Let $\mathbf{M}$ be an $\mathbf{L}'$-structure. Then for any $\mathbf{L}'$-sentence ϕ, $\models_{\mathbf{M}} \phi \Leftrightarrow \models_{L(\mathbf{M})} L(\phi)$.

Proof: The induction hypothesis is the following. If $\psi(x^1_{v_1}, \ldots, x^n_{v_n})$ is an $\mathbf{L}'$-formula with free variables $x^1_{v_1}, \ldots x^n_{v_n}$ and $l_i \in I_{\mathbf{M}}(v_i)$ for all i, $1 \leq i \leq n$, then

$$\models_{\mathbf{M}} \psi(l_1, \ldots, l_n) \Leftrightarrow \models_{L(\mathbf{M})} L(\psi)(l_1, \ldots, l_n)$$

Taking $\psi = \phi$ completes the proof of the theorem. We shall show the inductive proof only for the first type of atomic formula in the list above. The proofs for the other cases are similar. Once we know that the result holds for atomic formulas, it is easy to show that it holds for all other formulas.

We therefore let $\psi(x_u, y_{u'})$ be the $\mathbf{L}'$-formula $t_1 \in_w t_2$ where w is of type $(\bigcirc, v)$

$$t_1 = \pi_{u_1,v}\pi_{u_2,u_1} \cdots \pi_{u_n,u_{n-1}}\pi_{u,u_n}(x_u)$$

and

$$t_2 = \pi_{u'_1,w}\pi_{u'_2,u'_1} \cdots \pi_{u'_m,u'_{m-1}}\pi_{u',u'_m}(y_{u'})$$

$L(t_1 \in_w t_2)$ was defined as $Q_{t_1}Q_{t_2}(\psi_{t_1} \wedge \psi_{t_2} \wedge (z_v \in z_w))$ where z_v and z_w are new variables. Let $l \in I(u)$ and $l' \in I(u')$. Then $\models_{\mathbf{M}} (t_1 \in_w t_2)(l, l')$ holds iff

$$\Pi_v\Pi_{u_1} \cdots \Pi_{u_n}(l)\ (\in_w)_{\mathbf{M}}\ \Pi_w\Pi_{u'_1} \cdots \Pi_{u'_m}(l')$$

Let $l_v = \Pi_v\Pi_{u_1} \cdots \Pi_{u_n}(l)$ and $l_w = \Pi_w\Pi_{u'_1} \cdots \Pi_{u'_m}(l')$. Then $l_v(\in_w)_{\mathbf{M}} l_w$ and therefore $l_v \in l_w$. By their definition, there must be a sequence of l-values $l_{u_1}, \ldots, l_{u_n}, l_{u'_1}, \ldots, l_{u'_m}$ satisfying

$$\models_{L(\mathbf{M})} \Big(\psi_{t_1} \wedge \psi_{t_2} \wedge (z_v \in z_w)\Big)(l_v, l_{u_1}, \ldots, l_{u_n}, l_w, l_{u'_1}, \ldots, l_{u'_m}, l, l')$$

This implies that

$$\models_{\mathbf{M}} \psi(l, l') \Leftrightarrow \models_{L(\mathbf{M})} \Big(Q_{t_1}Q_{t_2}\Big(\psi_{t_1} \wedge \psi_{t_2} \wedge (z_v \in z_w)\Big)\Big)(l, l')$$

and therefore

$$\models_{\mathbf{M}} \psi(l, l') \Leftrightarrow \models_{L(\mathbf{M})} L(t_1 \in_w t_2)(l, l')$$ ∎

5.2.3. Consequences of the Reduction

It follows immediately from the definitions together with Lemma 5 that L and F are inverse mappings on instances.

Lemma 9:

1. If $\mathbf{I} = \langle I, r, f \rangle$ is an instance of $\mathbf{S}$, then $L(F(\mathbf{I})) = \mathbf{I}$.
2. If $\mathbf{M}$ is an $\mathbf{L}'$-structure that satisfies $\mathrm{Ax}(\mathbf{S})$, then $F(L(\mathbf{M})) = \mathbf{M}$. ∎

As functions on formulas, ϕ and ψ are not inverses, since $F(L(\phi))$ may be a different sentence from ϕ. However these sentences are logically equivalent.

Lemma 10:

1. Let ϕ be an $\mathbf{L}$-sentence. Then $L(F(\phi))$ is equivalent in $\mathbf{L}$ to ϕ.
2. Let ϕ be an $\mathbf{L}'$-sentence. Then $\mathrm{Ax}(\mathbf{S}) \vdash (F(L(\phi)) \Leftrightarrow \phi)$.

Proof:

1. We have to show that for any instance $\mathbf{I}$ of $\mathbf{S}$, $\models_{\mathbf{I}} \left(\phi \Leftrightarrow L(F(\phi))\right)$. By Theorem 6, $\models_{\mathbf{I}} \phi$ is equivalent to $\models_{F(\mathbf{I})} F(\phi)$, and by Theorem 8, $\models_{F(\mathbf{I})} F(\phi)$ is equivalent to $\models_{L(F(\mathbf{I}))} L(F(\phi))$. Finally, by Lemma 9, $L(F(\mathbf{I})) = \mathbf{I}$.
2. Let $\mathbf{M}$ be an $\mathbf{L}'$-structure satisfying $\mathrm{Ax}(\mathbf{S})$. By Theorem 8, $\models_{\mathbf{M}} \phi$ is equivalent to $\models_{L(\mathbf{M})} L(\phi)$. Theorem 6 implies that $\models_{L(\mathbf{M})} L(\phi)$ is equivalent to $\models_{F(L(\mathbf{M}))} F(L(\phi))$ and by Lemma 9, $F(L(\mathbf{M})) = \mathbf{M}$. Therefore $\models_{\mathbf{M}} F(L(\phi))$ is equivalent to $\models_{\mathbf{M}} \phi$, and therefore $\mathrm{Ax}(\mathbf{S}) \vdash (F(L(\phi)) \Leftrightarrow \phi)$. ∎

Corollary 11: (Validity) Let ϕ be an LDM sentence over $\mathbf{S}$. Then ϕ is valid if and only if $\mathrm{Ax}(\mathbf{S}) \vdash F(\phi)$.

Proof: Assume ϕ is valid. Let $\mathbf{M}$ be an $\mathbf{L}'$-structure satisfying $\mathrm{Ax}(\mathbf{S})$. Since ϕ is valid, $\models_{L(\mathbf{M})} \phi$. By Theorem 6, $\models_{F(L(\mathbf{M}))} F(\phi)$, and therefore $\models_{\mathbf{M}} F(\phi)$. This shows that $\mathrm{Ax}(\mathbf{S}) \vdash F(\phi)$. The proof of the converse is similar. ∎

Corollary 12: (Compactness) Let Σ be a set of LDM sentences over $\mathbf{S}$. Then Σ is satisfiable iff every finite subset of Σ is satisfiable.

Proof: Let $F(\Sigma) = \{F(\sigma) \mid \sigma \in \Sigma\}$. If $\mathbf{I}$ satisfies a finite subset of Σ, then by Theorem 6 $F(\mathbf{I})$ will satisfy the corresponding subset of $F(\Sigma)$. This shows that every finite subset of $F(\Sigma)$ is satisfiable by a model of $\mathrm{Ax}(\mathbf{S})$. The Compactness Theorem for first-order logic then implies that $F(\Sigma) \cup \mathrm{Ax}(\mathbf{S})$ is satisfiable by some model $\mathbf{M}$. By Theorem 8, all the sentences in $L(F(\Sigma))$ hold in $L(\mathbf{M})$, and by Lemma 10 the sentences in $L(F(\Sigma))$ are logically equivalent to those in Σ. ∎

Corollary 13: (Löwenheim-Skolem) Let Σ be a set of LDM-sentences over a schema $\mathbf{S}$. If Σ is satisfiable, then it is satisfiable by a countable instance.

Proof: The proof is similar to the proof of the Compactness Theorem, together with the observation that the mapping L preserves the cardinality of the model. ∎

While the latter two corollaries are of theoretical interest, the Validity Corollary also has a practical significance. It implies that together with the appropriate interface we can use a standard theorem-prover in the database design process or for deductive query processing [BBG78] [MMSU81] [NG78] [Rei84].

5.3. A Proof Theory for LDM Logic

In this section we give a complete set of axioms and derivation rules for LDM logic. The axioms are as follows.

1. All instances of propositional tautologies.
2. Logical axioms, as in first-order logic.
   (a) $\vdash (\forall x_v)(\phi \Rightarrow \psi) \Rightarrow ((\forall x_v)\phi \Rightarrow (\forall x_v)\psi)$
   (b) $\vdash (\forall x_v)\phi(x_v) \Rightarrow \phi(y_v)$, where y_v does not appear bound in ϕ.
3. Equality axioms for $=_l$.
   (a) $\vdash (\forall x_v)(x_v =_l x_v)$
   (b) $\vdash x_v =_l y_v \Rightarrow (\phi \Rightarrow \psi)$, where ψ is obtained from ϕ by replacing some or all occurrences of x_v by y_v.
4. Axioms that say that $=_r$ is an equivalence relation. If u, v, and w are nodes of **S** of type $\Box$, then the following are axioms.
   (a) $\vdash (x_u =_r x_u)$
   (b) $\vdash (x_u =_r y_v \Rightarrow y_v =_r x_u)$
   (c) $\vdash (x_u =_r y_v \wedge y_v =_r z_w \Rightarrow x_u =_r z_w)$
5. Axioms for $\bigcirc$-nodes. If u is of type $\bigcirc$ and v is its t^{th} child, then we have axioms saying that each l-value in $I(v)$ has a unique t^{th} projection.
   (a) $\vdash (\forall x_u)(\exists y_v)(y_v\ \pi_t\ x_u)$ (Existence).
   (b) $\vdash (\forall x_u)(\forall y_v)(\forall z_v)(y_v\ \pi_t\ x_u \wedge z_v\ \pi_t\ x_u \Rightarrow y_v =_l z_v)$ (Uniqueness).
6. Axioms for $\triangle$-nodes. If u is of type $(\triangle, n, v_1, \ldots, v_n)$, then there is exactly one element of the $I(v_i)$'s that corresponds to each element of u.
   (a) $(\forall x_u)\big((\exists y^1_{v_1})(y^1_{v_1}\ \rho\ x_u) \vee \cdots \vee (\exists y^n_{v_n})(y^n_{v_n}\ \rho\ x_u)\big)$ (Existence).
   (b) For all i, j where $1 \le i, j \le n$ and $i \ne j$,
   $$(\forall x_u)\big((\exists y^i_{v_i})(y^i_{v_i}\ \rho\ x_u) \Rightarrow (\forall y^j_{v_j})\neg(y^j_{v_j}\ \rho\ x_u)\big)$$
   (Uniqueness of the node among the children of u).
   (c) For all i, $1 \le i \le n$,
   $$(\forall x_u)(\forall y^1_{v_i})(\forall y^2_{v_i})((y^1_{v_i}\ \rho\ x_u) \wedge (y^2_{v_i}\ \rho\ x_u) \Rightarrow (y^1_{v_i} =_l y^2_{v_i}))$$
   (Uniqueness in that child).

The derivation rules are the same as in first-order logic, namely

(MP) From $\vdash \phi \Rightarrow \psi$ and $\vdash \phi$ we can infer $\vdash \psi$.

(Gen) From $\vdash \phi$ we can infer $\vdash (\forall x_v)\phi$ for any sort $v \in V$.

We use the standard notation for implication. Therefore $\Sigma \vdash \phi$ means that ϕ follows from Σ and the above axioms and derivation rules. We now show that this is a complete set of axioms.

Theorem 14:

$$\Sigma \vdash \phi \Leftrightarrow \Sigma \models \phi$$

Proof: We first prove that $\Sigma \models \phi \Leftrightarrow F(\Sigma) \models F(\phi)$. If $\Sigma \models \phi$, and $\mathbf{M}$ is a model of $F(\Sigma)$ satisfying Ax(S), then $L(\mathbf{M})$ is a model of $L(F(\Sigma))$. By Lemma 10, $L(\mathbf{M})$ satisfies Σ, and therefore satisfies ϕ. But then $\mathbf{M} = F(L(\mathbf{M}))$ satisfies $F(\phi)$ by Theorem 6. The proof of the converse is similar.

By the completeness of first-order logic, $F(\Sigma) \models F(\phi)$ is equivalent to $F(\Sigma) \vdash F(\phi)$. To complete the proof, it therefore remains to show that

$$\Sigma \vdash \phi \Leftrightarrow F(\Sigma) \vdash F(\phi)$$

To prove that $\Sigma \vdash \phi \Rightarrow F(\Sigma) \vdash F(\phi)$, we show that each axiom of the LDM logic $\mathbf{L}$ is mapped by F into a theorem of $\mathbf{L}'$, and that the derivation rules are mapped into valid rules.

It is easy to see that tautologies are mapped to tautologies. The other logical axioms are similar, e.g.,

$$(\forall x_v)(\phi \Rightarrow \psi) \Rightarrow \Big((\forall x_v)\phi \Rightarrow (\forall x_v)\psi\Big)$$

is mapped by F into

$$(\forall x_v)\Big(F(\phi) \Rightarrow F(\psi)\Big) \Rightarrow \Big((\forall x_v)F(\phi) \Rightarrow (\forall x_v)F(\psi)\Big)$$

which is valid in first-order logic.

The axioms for $=_l$ are similarly mapped into equality axioms of first-order logic. As for the axioms for $=_r$, $F(x_u =_r y_v \Rightarrow y_v =_r x_u)$, for example, is

$$(f_u(x_u) = f_v(y_v) \Rightarrow f_v(y_v) = f_u(x_u))$$

which is clearly valid.

The axioms for $\supset$ are mapped to the valid LDM sentences

$$(\forall x_u)(\exists y_v)(y_v = \pi_{u,i}(x_u))$$

and

$$(\forall x_u)(\forall y_v)(\forall z_v)(y_v = \pi_{u,i}x_u \wedge z_v = \pi_{u,i}x_u \Rightarrow y_v = z_v)$$

and the axioms for $\triangle$ are mapped into axioms in Ax(S). The proof that the derivation rules are valid is straightforward.

We shall now show that $\Sigma \vdash \phi \Rightarrow L(\Sigma) \vdash L(\phi)$. Once this holds, we then have $F(\Sigma) \vdash \phi \Rightarrow L(F(\Sigma)) \vdash L(F(\phi))$, and applying Lemma 10 completes the proof.

In order to prove this, we show that all the axioms of the first-order theory $\mathbf{L}'$ are mapped by L into consequences of the LDM axioms, and that the derivation rules are mapped into valid derivation rules. For this we need a set of axioms for many-sorted logic. Such a set of axioms consists [Sch38] of the standard first-order axioms with the obvious restrictions of sorts of variables and terms.

The proof for the derivation rules and equality axioms is straightforward. It is also straightforward to show that the axioms in Ax(S) are mapped into the $\triangle$-axioms, and that an instance of the logical axiom

$$\vdash (\forall x_v)(\phi \Rightarrow \psi) \Rightarrow ((\forall x_v)\phi \Rightarrow (\forall x_v)\psi)$$

is mapped into the corresponding LDM axiom.

The remaining, and most difficult, case is the logical axiom

$$\vdash (\forall x_v)\phi(x_v) \Rightarrow \phi(t)$$

where t is a term that contains no variables that are quantified in ϕ by a quantifier that has a free occurrence of x_v in its range. This is mapped into the formula $\vdash (\forall x_v)L(\phi(x_v)) \Rightarrow L(\phi(t))$. This might appear to be an instance of the corresponding LDM axiom, but it is not. The reason for this is that substituting t and then applying L does not give the same formula as applying L and then substituting t for x_v.

Let $\overline{\phi}$ be the result of substituting the term t for x_v in ϕ. We shall prove that $(\forall x_v)(L(\phi)) \Rightarrow L(\overline{\phi})$ is a theorem of LDM logic by showing, by induction on the size of ϕ, that the stronger assertion $(\forall x_v)(L(\phi)) \Rightarrow L(\overline{\phi}) \Rightarrow (\exists x_v)(L(\phi))$ is such a theorem. The proofs of all of the cases except when ϕ is atomic are trivial. Note that the second implication is needed for the proof of the first implication to go through in the case of negation.

For the case when ϕ is atomic, note first that v cannot be of sort $\overline{c}$, since there are no variables of this sort. The treatment of the various types of atomic formulas are all similar, and we shall prove the result for the case when ϕ is the formula $x_v \in y_w$. t must be a term of the form $\pi_v \pi_{v_1} \cdots \pi_{v_n}(z_u)$. $(\forall x_v)L(\phi)$ is then the formula

$$(\forall x_v)(x_v \in_w y_w)$$

$L(\overline{\phi})$ is the formula

$$(\exists x_v)(\exists x_{v_1}) \cdots (\exists x_{v_n})(x_v\ \pi_v\ x_{v_1} \wedge \cdots \wedge x_{v_n}\ \pi_{v_n}\ z_u \wedge x_v \in_w y_w)$$

and $(\exists x_v)L(\phi)$ is $(\exists x_v)(x_v \in_w y_w)$. Proving the induction hypothesis is now straightforward since the LDM axioms for $\frown$-nodes imply that for each z_u there are $x_{v_n}, \ldots, x_{v_1}, x_v$ satisfying

$$(x_v\ \pi_v\ x_{v_1} \wedge \cdots \wedge x_{v_n}\ \pi_{v_n}\ z_u)\ \blacksquare$$

Corollary 15: The axiom system introduced in this section is sound and complete for LDM logic. ∎

5.4. The Complexity of Integrity Checking

From now on we consider only instances that correspond to real databases. In other words all instance are finite, all l-values are natural numbers, all r-values in nodes of type □ are from a fixed set D, and we do not distinguish between individual constants in the schemas and data elements.

In this section we investigate the complexity of checking integrity constraints. The integrity constraints are sentences in LDM-logic, and a database is "legal" if and only if it satisfies the constraints. Following [Var82], we use two measures of complexity, *data complexity* and *expression complexity*. Intuitively, data complexity is the complexity of testing satisfaction of a fixed sentence in terms of the size of the database. Expression complexity, on the other hand, is the complexity of testing satisfaction of sentences on a fixed database in terms of the length of the sentences.

More formally, the data complexity of LDM logic is the complexity of the sets

$$Gr(\mathbf{S}, \phi) = \{\mathbf{I} \mid \mathbf{I} \text{ is an instance of } \mathbf{S} \text{ and } \models_{\mathbf{I}} \phi\}$$

where ϕ is a sentence over $\mathbf{S}$. The expression complexity of LDM logic is the complexity of the sets

$$Gr'(\mathbf{S}, \mathbf{I}) = \{\phi \mid \models_{\mathbf{I}} \phi\}$$

where $\mathbf{I}$ is an instance of $\mathbf{S}$. Note that $Gr(\mathbf{S}, \phi)$ is a set of instances, while $Gr'(\mathbf{S}, \mathbf{I})$ is a set of sentences.

Theorem 16:

1. For every sentence ϕ over $\mathbf{S}$, the set $Gr(\mathbf{S}, \phi)$ is in LOGSPACE.
2. For every instance $\mathbf{I}$ of $\mathbf{S}$, the set $Gr'(\mathbf{S}, \mathbf{I})$ is in PSPACE.
3. There is a schema $\mathbf{S}$ and an instance $\mathbf{I}$ of $\mathbf{S}$, such that the set $Gr'(\mathbf{S}, \mathbf{I})$ is logspace complete in PSPACE.

Proof:

1. We have to test whether, for a fixed sentence ϕ, $\models_{\mathbf{I}} \phi$. Let $|I| = n$ be the number of l-values in $\mathbf{I}$, and let k be the number of quantifiers in ϕ. In order to test whether $\models_{\mathbf{I}} \phi$, we have to test all possible assignments of values to these variables, of which there are at most n^k. If we cycle through these assignments in a fixed, say lexicographic, order, we can do this in space $O(k \log n) = O(\log n)$. For each such assignment it is easy to see that testing ϕ takes constant space.

2. As in the previous case, we can test $\models_{\mathbf{I}} \phi$ in $O(k \log n) = O(k)$ space. In this case n is fixed and k, the number of quantifiers in ϕ, is less than the length of ϕ.

3. We shall reduce the Quantified Boolean Formulas (QBF) of [Sto77] to the set $Gr'(\mathbf{S}, \mathbf{I})$. Let $\mathbf{S}$ be the schema consisting of the single node u of type $\square$. Let $\mathbf{I}$ be the instance of $\mathbf{S}$ with $I(u) = \{1, 2\}$ and $r(1) = F$, $r(2) = T$. If $E' = (Q_1 x_1) \cdots (Q_n x_n) E$ is an instance of QBF, let $\phi(E')$ be the LDM formula that we get by replacing each literal x_i in E by $x_u^i =_r T$, each $\overline{x}_u^i$ by $x_u^i =_r F$, and each quantifier $(Q_i x_i)$ by $(Q_i x_u^i)$. We clearly have $|\phi(E')| = c\,|E'|$, and also

$$\phi(E') \in Gr'(\mathbf{S}, \mathbf{I}) \Leftrightarrow \models_{\mathbf{I}} \phi(E')$$

$$\Leftrightarrow \text{There exists a satisfying truth assignment for } E' \blacksquare$$

Thus the data complexity of LDM logic is LOGSPACE, and the expression complexity of LDM logic is PSPACE. Since analogous results hold for the relational model [Var82], we see that integrity checking in the logical data model is not more difficult than in the relational model.

Chapter 6

The Logical Query Language

6.1. Introduction

In this chapter we use the LDM logic described in the previous chapter to define a non-procedural query language on LDM schemas. This language will be analogous to the tuple calculus in the relational model. As we mentioned earlier non-procedural languages exist for the relational model but not for the other models, and these models can only be queried through various procedural languages. For the rest of this thesis we consider only instances that correspond to real databases. In other words all instance are finite, all l-values are natural numbers, all r-values in nodes of type □ are from a fixed set D, and we do not distinguish between individual constants in the schemas and data elements. Throughout this chapter **S** will be a fixed schema. Except where mentioned otherwise, **I** will be a fixed instance of **S**.

We noted one major difference between the relational model and other models, namely that the result of a query in the relational model has the same structure as the relations in the database. This is certainly not true of most of the other data models. Whatever the result of a query on a hierarchical database is, using the standard query languages, it will not be another hierarchy. Because of this property the relational query language can be used for defining views, rather than requiring a separate language for view definition. Furthermore, the fact that the result of a query has the same structure as the database enables us to express and answer complex queries. The system can then break queries up into simpler subqueries and answer the simpler queries first.

We would therefore like the LDM queries to have a structure that is similar to that of the database, i.e., they should also be LDM schemas. Chapter 3 gives some idea of the sort of queries we should like to write.

The natural analogue to the relational calculus would be to have the query consist of an LDM formula ϕ containing one free variable for each query node. Intuitively, we should select all objects that satisfy the formula. This approach turned out not to work for several reasons. One was the difficulty of handling cyclic queries, while the other was what to do with nodes of type ○. The only way we were able to deal with ○ nodes was to require the query to group together as much as possible in each set. This both reduced the expressive power of such nodes, as we could no longer relate an object to more than one set, and also resulted in an extremely complicated and unintuitive definition of the result of the query.

Another unsuccessful approach, using a closed formula ϕ is described in Appendix A. The successful approach was base on the following idea. Suppose the query added just one node u to the schema. Then we could use a formula $\phi_u(x_u)$ with one free variable x_u of sort u to define explicitly what the contents of u are in terms of the contents of the database. The bound variables of $\phi_u(x_u)$ therefore can range over nodes of the schema **S**. The result of the query will be an extension of **I** such that u contains all those "objects" that satisfy ϕ.

What do we do if $\mathbf{Q}$ adds more than one node to $\mathbf{S}$? We decided to extend this approach by having one formula *per node*. Each such formula will define the result at its node in terms of the contents of the database and of nodes whose result has already been constructed. A consequence of this is that the query schema must be acyclic. As we still allow the database schema to be cyclic and only prevent the user from constructing new cycles in his queries we do not think that this is too serious a restriction.

6.2. The LDM Query Language

Definition 23: Let $\mathbf{S} = \langle V, E, \mu \rangle$ be an LDM schema. A query on $\mathbf{S}$ consists of a tuple $\mathbf{Q} = \langle \mathbf{S_Q}, \Phi_\mathbf{Q}, \prec_\mathbf{Q} \rangle$ where

1. $\mathbf{S_Q}$ is an extension of $\mathbf{S}$.
2. $\prec_\mathbf{Q}$ is a topological order on the nodes in $V_\mathbf{Q} - V$, i.e., $\prec_\mathbf{Q}$ is a linear order such that if v is a child of w then $v \prec_\mathbf{Q} w$.
3. $\Phi_\mathbf{Q}$ is a set of LDM formulas, one for each node v in $V_\mathbf{Q} - V$. The formula ϕ_v that corresponds to the node v satisfies
   (a) ϕ_v has only one free variable, and it is of sort v.
   (b) All other variables in ϕ_v are bound. Each of their sorts is either a node of the database schema $\mathbf{S}$ or is a query node that precedes v under $\prec_\mathbf{Q}$.

The order $\prec_\mathbf{Q}$ is used to specify the order in which we define the result of the query. In Section 6.4 we investigate to what extent we can do without this order.

Before continuing with the formal details we give several examples of logical queries. The database schema in these examples will be the genealogy schema of Fig. 8 (page 10). The instance of it will be that shown in Fig. 10 (page 11).

Example 12: The schema of $\mathbf{Q}_1$ is shown in Fig. 22, The formula $\phi_{u'}(x_{u'})$ is $(\exists y_u)(x_{u'} =_r y_u)$. In other words we want $I(u')$ to be a copy of $I(u)$. We eliminate, however, any duplication that may be in $I(u)$. The result of the query[1] is shown in Fig. 23.

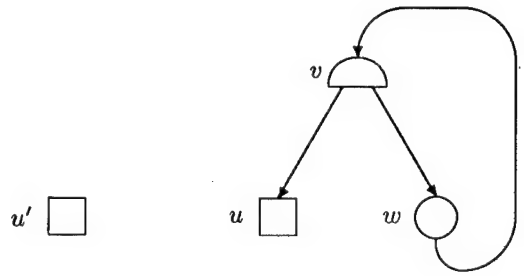


Figure 22: Schema of $\mathbf{Q}_1$

[1] In all these examples, the result is defined only up to isomorphism relative to $\mathbf{S}$, i.e., the choice of l-values is arbitrary

$I(u')$

l	$r(l)$
17	Rehoboam
18	Solomon
19	David
20	Batsheba
21	Jesse

Figure 23: Result of $\mathbf{Q}_1$

Example 13: The schema of $\mathbf{Q}_2$ is shown in Fig. 24. $\mathbf{Q}_2$ has $\phi_{u'}(x_{u'})$ always true. The result is quite large containing $2^6 = 64$ elements, the l-values 17 to 80. For this reason we do not show it here. The r-values of these l-values are all the subsets of $I(u)$.

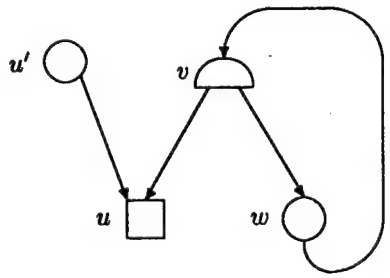


Figure 24: Schema of $\mathbf{Q}_2$

Example 14: The schema of $\mathbf{Q}_3$ is shown Fig. 25. We want v' to contain the set of parents of Solomon and so we have the formulas

$$\phi_{u'}(x_{u'}) = (\exists y_u^1)(\exists y_u^2)(\exists z_v^1)(\exists z_v^2)(\exists z_w^3)\Big((y_u^1 =_r x_{u'}) \wedge (y_u^2 =_r \text{“Solomon”}) \wedge (z_v^2 =_r (y_u^2, z_v^3)) \wedge (z_v^1 \in z_v^3) \wedge (y_u^1\ \pi_1\ z_v^1)\Big)$$

and $\phi_{v'}(x_{v'}) = (\forall y_{u'})(y_{u'} \in x_{v'})$.

What $\phi_{u'}(x_{u'})$ says is that there is some l-value (y_u^2) in $I(u)$ with the r-value “Solomon,” and another (y_u^1) with r-value equal to $x_{u'}$. The rest of the formula says that y_u^1 is a parent of y_u^2. $\phi_{v'}(x_{v'})$ says that $I(v')$ contains all the l-values in $I(u')$ in one set.
The result of the query is shown in Fig. 26.

Example 15: The schema of $\mathbf{Q}_4$ is shown in Fig. 27. We want to restructure the hierarchy as a relation, i.e., we want $I(v')$ and $I(w')$ to contain all the names of people that are in the database and $I(u')$ to connect people to their parents.
The formulas are

$$\phi_{v'}(x_{v'}) = (\exists y_u)(x_{v'} =_r y_u)$$

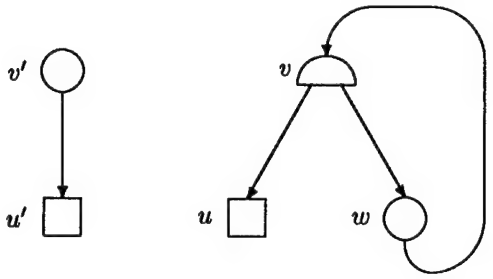


Figure 25: Schema of $\mathbf{Q}_3$

$I(u')$

l	$r(l)$
17	David
18	Batsheba

$I(v')$

l	$r(l)$
19	$\{17, 18\}$

Figure 26: Result of $\mathbf{Q}_3$

$$\phi_{w'}(x_{w'}) = (\exists y_u)(x_{w'} =_r y_u)$$

and

$$\phi_{u'}(x_{u'}) = (\exists x^1_{v'})(\exists x^2_{w'})(\exists y^1_u)(\exists y^2_u)(\exists z^1_v)(\exists z^2_v)(\exists z^3_w)\Big((x^1_{v'} =_r y^1_u) \wedge (x^2_{w'} =_r y^2_u) \wedge (x^1_{u'} =_r (x^1_{v'}, x^2_{w'})) \wedge (z^1_v =_r (y^1_u, z^3_w)) \wedge (y^2_u\ \pi_1\ z^2_v) \wedge (z^2_v \in z^3_w)\Big)$$

The result of the query is shown in Fig. 28.

We now formally define the result of a logical query. We start by looking at queries that add just one node to the schema. We shall call queries like this *simple queries.*

Definition 24: A query $\mathbf{Q}$ is called a *simple query* if $\left|V_{\mathbf{Q}} - V\right| = 1$. We shall use the notation $\mathbf{Q}_v$ for a simple query that has $V_{\mathbf{Q}} - V = \{v\}$.

Let $\mathbf{Q}_v$ be a simple query on a schema $\mathbf{S}$ and let $\mathbf{I}$ be an instance of $\mathbf{S}$. The result of $\mathbf{Q}_v$ on $\mathbf{I}$ will be an extension $\mathbf{I}_v$ of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$. In order to define $\mathbf{I}_v$ we have to define what $I_v(v)$ is and what the r-values of these l-values are. It should contain all those "objects" that satisfy $\phi_v(x_v)$. The problem with using this as a definition of $\mathbf{I}_v$ is that $\phi_v(x_v)$ is satisfied by *l-values* and since $I_v(v)$ has not yet been defined it is meaningless to talk about the objects that satisfy ϕ_v. It might seem that problem is trivial, but suppose that $\phi_v(x_v)$ included the conjunct $(\forall y_v)(\forall z_v)(y_v =_l z_v)$. In other words $I(v)$ can contain at most one l-value. If the rest of ϕ_v allowed several possibilities for the r-value of this l-value we would have no way of choosing which one would be in the result.

What enables us to deal with this problem is that a formula like this is not allowed in our query language—all bound variables in our language must refer to database nodes or nodes that precede v, not to v itself. As a result of this restriction, it will turn out that although ϕ_v refers to l-values, it really expresses something about their r-values alone. This will enable us to find the r-values that satisfy ϕ_v and after that pick the l-values arbitrarily.

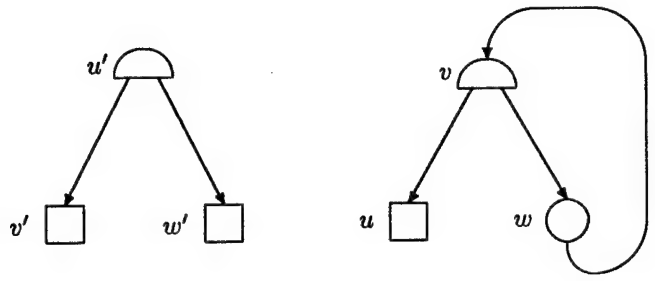


Figure 27: Schema of $\mathbf{Q}_4$

$I(v')$

l	$r(l)$
17	Rehoboam
18	Solomon
19	David
20	Batsheba
21	Jesse

$I(w')$

l	$r(l)$
22	Rehoboam
23	Solomon
24	David
25	Batsheba
26	Jesse

$I(u')$

l	$r(l)$
27	(17, 23)
28	(18, 24)
29	(18, 25)
30	(19, 26)

Figure 28: Result of $\mathbf{Q}_4$

Definition 25: Let r be an r-value (i.e., anything that could be an r-value of x_v). We say that r is a *candidate r-value for* v[2] if the following holds. Let l be some new l-value, i.e., one that does not appear in $\mathbf{I}$. Let $\mathbf{I}_v$ be the extension of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$ with $I(v) = \{l\}$ and $r(l) = r$. Then $\models_{\mathbf{I}} \phi_v(l)$.

By using this arbitrary l-value we are able to express the fact that r is one of objects that should be in the result of the query. We first show that the particular choice of l-value is unimportant.

Lemma 17: Let r be an r-value and let $\mathbf{I}_1$ and $\mathbf{I}_2$ be two extensions of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$ defined by, respectively, $I_1(v) = \{l_1\}$, $r_1(l_1) = r$, and $I_2(v) = \{l_2\}$, $r_2(l_2) = r$. Then $\models_{\mathbf{I}_1} \phi_v(l_1) \Leftrightarrow \models_{\mathbf{I}_2} \phi_v(l_2)$.

Proof: By definition ϕ_v has only one free variable of sort v, i.e., the variable x_v. By inspection, we can see that the only atomic formulas that can contain x_v are $x_w\ \pi_i\ x_v$, $x_w\ \rho\ x_v$, $x_w \in x_v$, $x_v =_r d$, $x_v =_r x_w$ and $x_v =_l x_v$. The last of these is always true, and it is easy to see that the truth of the others depends only on the r-value of x_v. The proof is then a straightforward induction. ∎

We now define the result of $\mathbf{S}_{\mathbf{Q}_v}$. Take all the candidate r-values for v, pick a new l-value for each one of them and put all of these l-values into $I_v(v)$. For now, we shall assume that the set of candidate r-values is finite. Queries with this property will correspond to the safe queries in the relational model. In the next section we shall look at this issue in more detail.

Definition 26: The result of $\mathbf{Q}_v$ is the extension $\mathbf{I}_v$ of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$ defined as follows. Let R be the set of all the candidate r-values for v and let $\{l_r \mid r \in R\}$ be a set of new l-values, i.e., ones that do not appear in $\mathbf{I}$. We then define $I_v(v)$ to be the set $\{l_r \mid r \in R\}$ and define $r(l_r) = r$ for each $r \in R$.

[2] Of course this really depends on $\mathbf{Q}$ and $\mathbf{I}$ as well, but these should be clear from the context.

We now show that this definition has the properties we want. We show that the result is well defined (up to isomorphism relative to $\mathbf{S}$, and assuming finiteness), that everything in the result satisfies ϕ_v and that we cannot add anything else that satisfies ϕ_v to the result without introducing duplication. Some of this formalizes what we meant when we said that ϕ_v expresses something about the r-values of x_v rather about than their l-values.

We first state a lemma which we shall need for the proof of Lemma 19. The proof of this lemma is similar to the proof of Lemma 17.

Lemma 18: Let $\mathbf{I}_1$ be an extension of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$. Let l be an element of $I_1(v)$ and let $\mathbf{I}_2$ be the extension of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$ defined by $\mathbf{I}_2(v) = \{l\}$ and $r_2(l) = r_1(l)$. Then $\models_{\mathbf{I}_1} \phi_v(l) \Leftrightarrow \models_{\mathbf{I}_2} \phi_v(l)$. ∎

Lemma 19:

1. Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be two results of $\mathbf{Q}_v$. Then $\mathbf{I}_1$ and $\mathbf{I}_2$ are isomorphic relative to $\mathbf{S}$.
2. Let $\mathbf{I}_v$ be the result of $\mathbf{S}_{\mathbf{Q}_v}$. Then for each l in $I_v(v)$, $\models_{\mathbf{I}_v} \phi_v(l)$.

Proof:

1. Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be two possible results of $\mathbf{Q}$ and let l_1 be an element of $I_1(v)$. Since $r(l_1)$ is a candidate r-value for v there must be some l_2 in $I_2(v)$ with $r(l_2) = r(l_1)$. Since both $I_1(v)$ and $I_2(v)$ have no duplication we immediately get a 1–1 correspondence between the l-values of $I_1(v)$ and $I_2(v)$. It is easy to see that this correspondence is an isomorphism.
2. Let l be an arbitrary element of $I_v(v)$, and let $\mathbf{I}^* = \langle I^*, r^* \rangle$ be the extension of $\mathbf{I}$ to $\mathbf{S}_{\mathbf{Q}_v}$ defined by $I^*(v) = \{l\}$ and $r^*(l) = r(l)$. By Lemma 18

$$\models_{\mathbf{I}_v} \phi_v(l) \Leftrightarrow \models_{\mathbf{I}^*} \phi_v(l)$$

Since $r(l)$ is a candidate r-value for v we can extend $\mathbf{I}$ to an instance $\mathbf{I}^{**}$ of $\mathbf{S}_{\mathbf{Q}_v}$ by defining $I^{**}(v) = \{l^{**}\}$, $r^{**}(l^{**}) = r(l)$, for some new l-value l^{**}. We then have $\models_{\mathbf{I}^{**}} \phi_v(l^{**})$. By Lemma 17, $\models_{\mathbf{I}^*} \phi_v(l)$ and therefore $\models_{\mathbf{I}} \phi_v(l)$. ∎

We now define the result of an arbitrary query $\mathbf{Q}$. To do this, we first define composition of queries.

Definition 27: Let $\mathbf{Q}_1$ be a query and let $\mathbf{Q}_2$ be a query on $\mathbf{S}_{\mathbf{Q}_1}$. $\mathbf{Q}_2 \circ \mathbf{Q}_1$ is the query on $\mathbf{S}$ that we get by composing them, i.e., $\mathbf{Q}_2 \circ \mathbf{Q}_1$ has $\mathbf{S}_{\mathbf{Q}_2 \circ \mathbf{Q}_1} = \mathbf{S}_{\mathbf{Q}_2}$, $\Phi_{\mathbf{Q}_2 \circ \mathbf{Q}_1} = \Phi_{\mathbf{Q}_1} \cup \Phi_{\mathbf{Q}_2}$ and

$$\prec_{\mathbf{Q}_2 \circ \mathbf{Q}_1} = \prec_{\mathbf{Q}_1} \cup \prec_{\mathbf{Q}_2} \cup \left\{(v, w) \mid v \in V_{\mathbf{Q}_1}, w \in V_{\mathbf{Q}_2}\right\}$$

Lemma 20: $\mathbf{Q}_2 \circ \mathbf{Q}_1$ is a query on $\mathbf{S}$. ∎

Let the nodes added by the query $\mathbf{Q}$ be $V_{\mathbf{Q}} - V = \{v_1, \ldots, v_n\}$ where $v_1 \prec \cdots \prec v_n$. We shall define a sequence of simple queries $\mathbf{Q}_{v_1}, \ldots, \mathbf{Q}_{v_n}$, as follows. Each $\mathbf{Q}_{v_i}$ is a query on the schema of $\mathbf{Q}_{v_{i-1}}$ and adds the node v_i to that schema. The formula for v_i is ϕ_{v_i}. It is easy to see that $\mathbf{Q} = \mathbf{Q}_{v_n} \circ \cdots \circ \mathbf{Q}_{v_1}$ and this enables us to easily define the result of $\mathbf{Q}$.

Definition 28: The result of the query $\mathbf{Q}$ on $\mathbf{I}$ is the result of applying the queries $\mathbf{Q}_{v_1}, \ldots, \mathbf{Q}_{v_n}$ successively to $\mathbf{I}$.

Lemma 21: The result of $\mathbf{Q}$ is well defined, i.e., different choices of l-values at each step yield isomorphic results.

Proof: This is a straightforward application of the first part of Lemma 19. ∎

The following theorem shows that the result of the query has the desired properties. These include a close relation with the maximize data while minimizing duplication approach that we described in the previous section.

Theorem 22: Let $\mathbf{I_Q}$ be the result of the query $\mathbf{Q}$ on the instance $\mathbf{I}$

1. Let v be a node added by $\mathbf{Q}$ and let l be an element of $I(v)$. Then $\models_{\mathbf{I}_v} \phi_v(l)$.
2. If v is a node added by $\mathbf{Q}$ and l_1 and l_2 are two different l-values in $I(v)$ then $r(l_1) \neq r(l_2)$. In other words there is no duplication in the result.
3. $\mathbf{I}_v$ is a maximal extension of $\mathbf{I}$ to $\mathbf{S_Q}$ that satisfies 1–2, i.e., there is no extension $\mathbf{I}_v^*$ with $I_v^*(v) \supseteq I_v(v)$ for all $v \in V_\mathbf{Q} - V$ that satisfies 1–2 and such that for at least one v the inclusion is proper.

Proof:

1. Let $\mathbf{Q}^*$ be the query $\mathbf{Q}_{v_k} \circ \cdots \circ \mathbf{Q}_{v_1}$ where $v = v_k$ and let $\mathbf{I_{Q^*}}$ be the result of $\mathbf{Q}^*$. By Lemma 19, $\models_{\mathbf{I_{Q^*}}} \phi_v(l)$. It is easy to see that $\mathbf{I}$ is an extension of an isomorphic image of $\mathbf{I_Q^*}$ and that extending $\mathbf{I_Q^*}$ to $\mathbf{S_Q}$ does not affect the satisfaction of ϕ_v.
2. Obvious.
3. Assume that such an $\mathbf{I}^*$ exists. Let $v = v_k$ be the first of the nodes $v_1, \ldots, v_n$ for which $I_v^*(v) \neq I_v(v)$ and let $\mathbf{Q}^*$ be the query $\mathbf{Q}_{v_k} \circ \cdots \circ \mathbf{Q}_{v_1}$. From 1 and 2 it follows immediately that both $\mathbf{I}_v$ and $\mathbf{I}_v^*$ restricted to $\mathbf{S_{Q^*}}$ are results of $\mathbf{Q}^*$. Lemma 21 then implies that $\mathbf{I}_v^*$ and $\mathbf{I}_v$ are isomorphic, a contradiction. ∎

6.3. Safe Queries

We have seen that provided that the set of candidate r-values at each node is finite, the result of the query is well-defined. It remains to see when the set of candidate r-values is finite.

Definition 29: A query $\mathbf{Q}$ on a schema $\mathbf{S}$ is *safe* if for every instance $\mathbf{I}$ of $\mathbf{S}$, the set of candidate r-values at each node, under the construction described above, is finite.

Note that as we are considering only finite instances, this is the same as requiring that the query have a result on every database instance.

Let v be a query node, i.e., an element of $V_\mathbf{Q} - V$. Assume that we have defined the result of $\mathbf{Q}$ for all those nodes that precede v. If $\mu(v) =$ ⌓, ○ or △, the set of candidate r-values for v is contained in either the cartesian product, union or powerset of the instance(s) of its child(ren) and therefore must be finite. The only case when it may be infinite is when $\mu(v) = \square$. If the domain D of data is finite, then all queries are safe, since the set of candidate r-values for nodes of type $\square$ is a subset of D. We therefore assume throughout this section that D is infinite.

Lemma 23: $\mathbf{Q}$ is safe on $\mathbf{I}$ iff for every query node of type $\square$ the set of candidate r-values for v is finite. ∎

We give two examples using the database and query schema shown in Fig. 22 (page 37) and the database instance shown in Fig. 10 (page 11).

Example 16: $\phi_{u'}(x_{u'})$ is $(\exists y_u)(x_{u'} =_r y_u) \vee (x_{u'} =_r$ "Absalom"). This query is safe since the set of candidate r-values is $R =$ { Jesse, David, Batsheba, Solomon, Rehoboam, Absalom}.

Example 17: $\phi_{u'}(x_{u'})$ is $(x_{u'} \neq_r$ "David"). This query is unsafe since the set of candidate r-values is $R = D -$ {David}, an infinite set.

As we have pointed out above, testing whether a relational query is safe is undecidable. As we can reduce testing safety of relational queries to testing safety of LDM queries there cannot be a decision procedure that tells us whether an query is safe on all database instances.

We give, however, a decision procedure for safety on fixed instances. Let **I** be a fixed instance of **S** and let **Q** be a query on **S**.

Lemma 24: Let $w_1, \ldots, w_n$ be all of the nodes in the schema **S** that are of type $\square$ and let $\{d_1, \ldots, d_k\}$ be the constants that occur in any of the query formulas. **Q** is safe on **I** iff for each query node v of type $\square$, every candidate r-value for v is either a) the r-value of an element of some $I(w_i)$ or b) one of the d_j's.

Proof: One direction is obvious—if this condition holds then **Q** is safe on **I**. We prove the converse by induction on the query nodes $V_{\mathbf{Q}} - V = \{v_1, \ldots, v_n\}$ where $v_1 \prec \cdots \prec v_n$. Let $v = v_i$ be a query node of type $\square$. We assume that the lemma holds for the nodes that precede v_i and that the query is safe on **I**. Let $\mathbf{I}_{i-1}$ be the result of $\mathbf{Q}_{v_{i-1}}$.

Since $\mathbf{Q}_v$ is safe on **I** the set of candidate r-values for v is a finite set R. We have to show that

$$R \subseteq \{d_1, \ldots, d_k\} \cup \bigcup_{\substack{\mu(w) = \square \\ w \in V}} I(w)$$

Call the right hand of this equation S. If the lemma is false, then there is some element r in $R - S$. By the induction hypothesis

$$S = \{d_1, \ldots, d_k\} \cup \bigcup_{\substack{\mu(w) = \square \\ w \in V \text{ or} \\ w \in V_{\mathbf{Q}}, w \prec v}} I_{i-1}(w)$$

Since r is a candidate r-value for v, if we extend $\mathbf{I}_{i-1}$ to an instance $\mathbf{I}_i^1$ of $\mathbf{S}_{\mathbf{Q}_v}$ by defining $I_i^1(v) = \{l\}$ and $r_1(l) = r$, we have $\models_{\mathbf{I}_i^1} \phi_v(l)$. Let r' be an arbitrary element of $D - S$, and extend $\mathbf{I}_{i-1}$ to an instance $\mathbf{I}_i^2$ of $\mathbf{S}_{\mathbf{Q}_v}$ by defining $I_i^2(v) = \{l\}$ and $r_2(l) = r'$. Since r and r' do not appear in the database, previously constructed nodes, or in the query formulas, an induction shows that

$$\models_{\mathbf{I}_i^1} \phi_v(l) \Leftrightarrow \models_{\mathbf{I}_i^2} \phi_v(l)$$

The key point in the induction is that x_v can occur in $\phi_v(x_v)$ only in atomic formulas of the form $x_v =_r d_j$ and $x_v =_r y_w$, where w is a node of type $\square$ that is either in V or is one of the nodes $v_1, \ldots, v_{i-1}$ The only other atomic formulas that can involve x_v are $x_v =_l x_v$ and $x_v =_r x_v$, and these are always true. All these formulas are false whenever the r-value of x_v is not in **S**.

We have therefore shown that all the elements of the infinite set $D - S$ are candidate r-values, a contradiction. ∎

The technique of this proof gives us an effective procedure for determining whether the simple query $\mathbf{Q}_v$ is safe on the instance **I**. Take some constant d_0 that does not occur anywhere in the database or in the query formulas. Test if d_0 is a candidate r-value (it is not difficult to see that this can be done effectively).

In a similar way to the proof of the above lemma, we can show that $\mathbf{Q}_v$ is safe on $\mathbf{I}$ iff d_0 is not a candidate r-value for v. Intuitively, if some such d_0 is in the result, the result is infinite since d_0 cannot be distinguished from any other such constant.

Combining this result with those of the previous section we get:

Theorem 25: Let $\mathbf{Q}$ be a query on $\mathbf{S}$ and let $\mathbf{I}$ be an instance of $\mathbf{S}$. There is a decision procedure to test whether $\mathbf{Q}$ is safe on $\mathbf{I}$. If $\mathbf{Q}$ is safe on $\mathbf{I}$ then the result can be computed effectively. ∎

Even though testing for safety and computing the result can be done effectively, it can still be NP-hard to do so, as we shall see in Section6.5.

6.4. Ordering the Nodes in a Query

We now examine more closely the role of the topological order in an LDM query. It might seem at first that we can relax the requirement. If each ϕ_v referred only to database nodes and to descendants of v, we could evaluate the query "bottom-up" without having to specify explicitly the evaluation order as part of the query.

Let us call the query language we would then get the *bottom-up query language*. The reason we prefer the LDM query language to the bottom-up query language is that the bottom-up language is not closed under composition.

The reason it is not is as follows. Let $\mathbf{Q}_2$ be a query on the result of $\mathbf{Q}_1$. Then the formula for a node v in $\mathbf{Q}_2$ can refer to a node u in $\mathbf{Q}_1$ that is not a descendant of v. This by itself does not necessarily mean that the language is not closed under composition—we might be able to rewrite the formula ϕ_v to get an equivalent query that does not refer to u. For example, if u is of type ⌓ we can rewrite ϕ_v to refer only to the descendants of u. We now show that if u is of type ○ this cannot always we done.

Theorem 26: The bottom-up query language is not closed under composition.

Proof: The database schema $\mathbf{S}$ consists of the node v in Fig. 29. $\mathbf{Q}_1$ adds the nodes u and w, and $\mathbf{Q}_2$ adds the node t to the result of $\mathbf{Q}_1$.

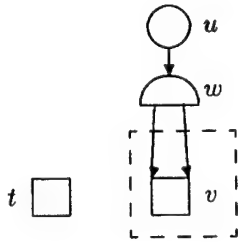


Figure 29: Query used in the proof of Theorem 26

The outline of the proof is as follows. We first show how by a suitable definition of $\mathbf{Q}_1$ and $\mathbf{Q}_2$ we can get $I(t)$ to contain copies of exactly those r-values in $I(v)$ that occur with the most duplication. If there were a bottom-up query equivalent to $\mathbf{Q}_2 \circ \mathbf{Q}_1$, it would have to define $I(t)$ in terms of database nodes and descendants of t, i.e. in terms of $I(v)$, alone. In the second part of the proof we show that this cannot be done.

Let $\mathbf{I}$ be an instance of $\mathbf{S}$ and let $d_1, \ldots, d_k$ be all the different r-values that occur in $I(v)$. Write $I(v)$ in the form

$$\{l_1^1, \ldots, l_{i_1}^1, l_1^2, \ldots, l_{i_2}^2, \ldots, l_1^k, \ldots, l_{i_k}^k\}$$

where $r(l_j^s) = d_j$. In other words, group the l-values in $I(v)$ by their r-values.

We define $\mathbf{Q}_1$ and $\mathbf{Q}_2$ by giving the formulas ϕ_w, ϕ_u and ϕ_t, and we show what the results of the queries are. $\phi_w(x_w)$ is the formula $(x_w =_l x_w)$ (or any other tautology). The result is simply the cross-product of $I(v)$ with itself, i.e., the candidate r-values for v are $\{(l_1, l_2) \mid l_1, l_2 \in I(v)\}$.

$\phi_u(x_u)$ is the formula

$$\begin{aligned}
& (\exists y_v^1)(\exists y_v^2)\Big((y_v^1 \neq_r y_v^2) \\
\wedge\ & (\exists z_w)\Big(z_w \in x_u \wedge z_w =_r (y_v^1, y_v^2)\Big) \\
\wedge\ & (\forall y_v^3)(\forall y_v^4)(\forall z_w)(z_w \in x_u \wedge z_w =_r (y_v^3, y_v^4) \Rightarrow y_v^3 =_r y_v^1 \wedge y_v^4 =_r y_v^2) \\
\wedge\ & (\forall z_w^1)(\forall z_w^2)(\forall y_v^3)(z_w^1 \in x_u \wedge z_w^2 \in x_u \wedge y_v^3\ \pi_1\ z_w^1 \wedge y_v^3\ \pi_1\ z_w^2 \Rightarrow z_w^1 =_l z_w^2) \\
\wedge\ & (\forall y_v^3)\Big(y_v^3 =_r y_v^1 \Rightarrow (\exists z_w)(z_w \in x_u \wedge y_v^3\ \pi_1\ z_w)\Big) \\
\wedge\ & (\forall z_w^1)(\forall z_w^2)(\forall y_v^3)(\forall y_v^4)(\forall y_v^5)(z_w^1 \in x_u \wedge z_w^2 \in x_u \wedge z_w^1 =_r (y_v^3, y_v^4) \\
& \wedge\ z_w^2 =_r (y_v^3, y_v^5) \Rightarrow y_v^4 =_l y_v^5)\Big)
\end{aligned}$$

$I(u)$ contains essentially all 1–1 functions from sets of the form $\{l_1^a, \ldots, l_{i_a}^a\}$ into $\{l_1^b, \ldots, l_{i_b}^b\}$ where $a \neq b$. More precisely, the candidate r-values for u are those $R \subseteq I(w)$ for which the set $r^* = \{r(l) \mid l \in R\}$ is such a function.

Let R be a candidate r-value and define r^* as above. Let l be a new l-value, and extend $\mathbf{I}$ to the node u by defining $I(u) = \{l\}$ and $r(l) = R$. Then $\models_{\mathbf{I}} \phi_u(l)$. Let l_i^a and l_j^b be l-values in $I(v)$ that correspond to the first two existential quantifiers in ϕ_u. By the first conjunct $a \neq b$. By the second conjunct $(l_i^a, l_j^b) \in R$. By the third, if $(l_1, l_2) \in r^*$ then $r(l_1) = a$ and $r(l_2) = b$. Therefore r^* is a subset of $\{l_1^a, \ldots, l_{i_a}^a\} \times \{l_1^b, \ldots, l_{i_b}^b\}$. The fourth conjunct implies that r^* is a function and the fifth that its domain is the entire set $\{l_1^a, \ldots, l_{i_a}^a\}$. Finally, the sixth conjunct implies that r^* is 1–1. In a similar way, given any such function r^* we can show that the set $\{l \in I(w) \mid r(l) \in r^*\}$ is a candidate r-value.

We now use these functions to find those r-values that occur in $I(v)$ with the most duplication. They are those d_a's for which there is a 1–1 function from each set $\{l_1^b, \ldots, l_{i_b}^b\}$ into the set $\{l_1^a, \ldots, l_{i_a}^a\}$. We formalize this by defining $\phi_t(x_t)$ as

$$\begin{aligned}
& (\exists y_v^1)\Big(y_v^1 =_r x_t \\
\wedge\ & (\forall y_v^2)(y_v^1 \neq_r y_v^2) \Rightarrow (\exists x_u)\Big((\exists z_w)(z_w \in x_u \wedge y_v^2\ \pi_1\ z_w) \\
\wedge\ & (\exists z_w)(\exists y_v^3)(z_w \in x_u \wedge y_v^3\ \pi_2\ z_w \wedge y_v^1 =_r y_v^3)\Big)\Big)
\end{aligned}$$

Let d be a candidate r-value for t. By the first conjunct in ϕ_t, d is one of the constants $d_1, \ldots, d_k$, say $d = d_a$. By the second conjunct, for any $d_b \neq d_a$ there is a 1–1 function from $\{l_1^b, \ldots, l_{i_b}^b\}$ to $\{l_1^a, \ldots, l_{i_a}^a\}$ and therefore d_a occurs in $I(v)$ with at least as much duplication as d_b. The converse is shown in a similar way.

To prove that the bottom-up query language is not closed under composition, it remains to show that there is no bottom-up query equivalent to $\mathbf{Q}_2 \circ \mathbf{Q}_1$. Such a query would have to define $I(t)$ by a formula $\phi_t(x_t)$ all of whose bound variables are all of sort v.

Let ϕ_t be such a formula and let n be the number of quantifiers it contains. Let $\mathbf{I}$ be an instance of $\mathbf{S}$ such that $I(u)$ contains $n+1$ copies of a, i.e., l-values with a as their r-value and $n+2$ copies of another

constant b. Let $\mathbf{I}^*$ be a second instance of $\mathbf{S}$, that differs from $\mathbf{I}$ only by containing another copy of a, i.e., another l-value l^* with r-value a. Then the only candidate r-values for t on $\mathbf{I}$ should be b and both a and b should be candidate r-values on $\mathbf{I}^*$.

Let l_t be a new l-value. Extend both $\mathbf{I}$ and $\mathbf{I}^*$ to t by defining $I(t) = I^*(t) = \{l_t\}$ and $r(l_t) = r^*(l_t) = a$. We shall complete the proof by showing that $\models_{\mathbf{I}} \phi_t(l_t) \Leftrightarrow \models_{\mathbf{I}^*} \phi_t(l_t)$ and thus contradicting the fact that a is a candidate r-value for t on $\mathbf{I}^*$ and not on $\mathbf{I}$.

Let the free variables of ϕ be among the variables $x_t, x_v^1, \ldots, x_v^n$. We prove the following by induction.

Let $l_{k+1}, \ldots, l_n$, or any other subset of the l_i's be elements of $I(v)$. Let $\tilde{l}$ be any element of $I(v)$ distinct from $l_{k+1}, \ldots, l_n$ that satisfies $r(\tilde{l}) = a$. Then

$$\models_{\mathbf{I}^*} \phi(l_t, l^*, \ldots, l^*, l_{k+1}, \ldots, l_n) \Leftrightarrow \models_{\mathbf{I}} \phi(l_t, \tilde{l}, \ldots, \tilde{l}, l_{k+1}, \ldots, l_n)$$

(Note that l^* is the l-value that appears in $\mathbf{I}^*$ but not in $\mathbf{I}$.) Using this with $\phi = \phi_t$ will complete the proof.

When ϕ is an atomic formula or $\neg\phi_1$ or $\phi_1 \wedge \phi_2$ the proofs of this assertion is straightforward. The hard case is when ϕ is $(\forall x_v^i)\psi(x_t, x_v^1, \ldots, x_v^n)$. Given a set $A \subseteq I(v)$ we shall say that $\tilde{l}$ is *suitable for* A if $r(\tilde{l}) = a$ and $\tilde{l}$ is not an element of A. If $\tilde{l}$ is suitable for A and some $l \in A$ has r-value a, it is easy to see that l is suitable for $A - \{l\} \cup \{\tilde{l}\}$. The assertion says that if $\tilde{l}$ is suitable for $\{l_{k+1}, \ldots, l_n\}$ then the equivalence holds. Since $I(v)$ has $n+1$ copies of a, we can always find at least two suitable l-values for any set A of size less than n.

To show the first direction, let $\tilde{l}$ be suitable for $\{l_{k+1}, \ldots, l_n\}$ and assume that

$$\models_{\mathbf{I}^*} ((\forall x_v^i)\phi)(l_t, l^*, \ldots, l^*, l_{k+1}, \ldots, l_n)$$

Since the value assigned to the quantified variable x_v^i is irrelevant, assume, w.l.o.g., that $i \geq k+1$. For all $l_0 \in I(v) \subseteq I^*(v)$, $\models_{\mathbf{I}^*} \phi(l_t, l^*, \ldots, l^*, l_{k+1}, \ldots, l_0, \ldots, l_n)$

1. If $l_0 \neq \tilde{l}$, then $\tilde{l}$ is suitable for $\{l_{k+1}, \ldots, l_n\}$. The induction hypothesis then implies that

$$\models_{\mathbf{I}} \phi(l_t, \tilde{l}, \ldots, \tilde{l}, l_{k+1}, \ldots, l_0, \ldots, l_n)$$

2. Replacing the quantified variable by l^*, we get

$$\models_{\mathbf{I}^*} \phi(l_t, l^*, \ldots, l^*, l_{k+1}, \ldots, l^*, \ldots, l_n)$$

Since $\tilde{l}$ is suitable for $\{l_{k+1}, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\}$ the induction hypothesis implies

$$\models_{\mathbf{I}} \phi(l_t, \tilde{l}, \ldots, \tilde{l}, l_{k+1}, \ldots, \tilde{l}, \ldots, l_n)$$

Combining these two, we get

$$\models_{\mathbf{I}} ((\forall x_v^i)\phi)(l_t, \tilde{l}, \ldots, \tilde{l}, l_{k+1}, \ldots, l_n)$$

For the converse, assume that $\models_{\mathbf{I}} ((\forall x_v^i)\phi)(l_t, \tilde{l}, \ldots, \tilde{l}, l_{k+1}, \ldots, l_n)$ holds. Then, for all $l_0 \in I(v)$

$$\models_{\mathbf{I}} \phi(l_t, \tilde{l}, \ldots, \tilde{l}, l_{k+1}, \ldots, l_0, \ldots, l_n)$$

1. If $l_0 \neq \tilde{l}$, then $\tilde{l}$ is suitable for $\{l_0, l_{k+1}, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\}$ and therefore

$$\models_{\mathbf{I}^*} \phi(l_t, l^*, \ldots, l^*, l_{k+1}, \ldots, l_0, \ldots, l_n)$$

2. If $l_0 = \tilde{l}$, then $\tilde{l}$ is suitable for $\{l_{k+1}, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\}$ and we get

$$\models_{\mathbf{I}^*} \phi(l_t, l^*, \ldots, l^*, l_{k+1}, \ldots, l^*, \ldots, l_n)$$

3. So far, we have shown that $\models_{\mathbf{I}^*} \phi(l_t, l^*, \ldots, l^*, l_{k+1}, \ldots, l_0, \ldots, l_n)$ for any l_0 in $I(v)$. Pick two l-values l_0 and l_0' that are both suitable for

$$\{\tilde{l}, l_{k+1}, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\}$$

Then l_0' is suitable for $\{l_0, l_{k+1}, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\}$ and the induction assumption implies that

$$\models_{\mathbf{I}} \phi(l_t, l_0', \ldots, l_0', l_{k+1}, \ldots, l_0, \ldots, l_n)$$

l_0 is suitable for $\{l_0', l_{k+1}, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\}$ and the induction hypothesis now gives us

$$\models_{\mathbf{I}^*} \phi(l_t, l_0', \ldots, l_0', l_{k+1}, \ldots, l^*, \ldots, l_n)$$

Using the induction assumption once more, together with the fact that $\tilde{l}$ is suitable for

$$\{l_0', l_{k+1}, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\}$$

gives us $\models_{\mathbf{I}} \phi(l_t, l_0', \ldots, l_0', l_{k+1}, \ldots, \tilde{l}, \ldots, l_n)$ Finally we use the induction hypothesis another time, this time with the fact that l_0' is suitable for

$$\{\tilde{l}', l_{k+1}, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n\}$$

to get $\models_{\mathbf{I}^*} \phi(l_t, l^*, \ldots, l^*, l_{k+1}, \ldots, \tilde{l}, \ldots, l_n)$.

Combining these shows that $\models_{\mathbf{I}^*} ((\forall x_v^i)\phi)(l_t, l^*, \ldots, l^*, l_{k+1}, \ldots, l_n)$ and completes the proof. ∎

As a consequence of this theorem we see that the topological order is a necessary part of the definition of the LDM query language. However, this does not mean that the user has to explicitly specify the order as part of the query, since it is enough if he just specifies the formulas at the nodes of the query. The system can then pick some order on the query nodes that is consistent with the graph edges and the implicit dependencies of one formula on another, i.e., if the formula for v refers to the node u then u must precede v. If the query is a legal one such an order must exist. The specific order we pick, subject to these constraints, turns out to be irrelevant. The following theorem shows that if we pick a different ordering we would get an equivalent query.

Theorem 27: Let $\mathbf{Q}_1 = \left\langle \mathbf{S}_\mathbf{Q}, \prec_1, \Phi \right\rangle$ and $\mathbf{Q}_2 = \left\langle \mathbf{S}_\mathbf{Q}, \prec_2, \Phi \right\rangle$ be two queries on a schema $\mathbf{S}$ that differ only in the topological order. Let $\mathbf{I}$ be an instance of $\mathbf{S}$. Then the results of $\mathbf{Q}_1$ and $\mathbf{Q}_2$ on $\mathbf{I}$ are isomorphic relative to $\mathbf{S}$.

Proof: Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be the two results. Let the query nodes be $V_\mathbf{Q} - V = \{v_1, \ldots, v_n\}$ where $v_1 \prec_1 \cdots \prec_1 v_n$. We define an isomorphism f from $\mathbf{I}_1$ to $\mathbf{I}_2$ by induction on the order $\prec_1$. Assume f has been defined for all w such that $w \prec_1 v$. Let R be the set of candidate r-values for v and write $I_1(v)$ as $\{l_r \mid r \in R\}$. We first define a mapping f^* on the candidate r-values for v as follows.

1. If $\mu(v) = \square$, then $f^*(r) = r$.

2. If $\mu(v) = (\supset\!\!\!\!\!\!\bigcirc, n)$, then r is a tuple $(l_1, \ldots, l_n)$ and we define

$$f^*(r) = \Big(f(l_1), \ldots, f(l_n)\Big)$$

3. If $\mu(v) = \triangle$ then r is the l-value l and we define $f^*(r) = f(l)$.

4. If $\mu(v) = \bigcirc$, then r is a set and we define $f^*(r) = \{f(l) \mid l \in r\}$.

It is not hard to show that $f^*(r)$ is a candidate r-value for v in $\mathbf{Q}_2$. Somewhat informally, the proof is as follows. Let $I_1(v)$ consist of the single l-value l_1 with r-value r and let $I_2(v)$ consist of the single l-value l_2 with r-value $f^*(r)$. Restrict the schema in both cases to the database $\mathbf{S}$ and those nodes that precede v in both $\mathbf{Q}_1$ and $\mathbf{Q}_2$. We can then show that by defining $f(l_1) = l_2$ we get an isomorphism between the instances. Theorem 3 then shows that $f^*(r)$ is a candidate r-value for v in $\mathbf{Q}_2$.

We can show in a similar way that the image of f^* contains all the candidate r-values for v in $\mathbf{Q}_2$, and we can therefore write $I_2(v)$ as $\{l_{f^*(r)} \mid r \in R\}$ where $r(l_{f^*(r)}) = f^*(r)$. By defining $f(l_r) = l_{f^*(r)}$ we get the desired isomorphism. ∎

6.5. Complexity of the Query Language

It is clear that the complexity of evaluating a query can be exponential, or worse, is the size of the database instance, since even the size of the result itself can be multiply exponential in the database size. We therefore ask the following question: Given a query and a database instance what is the complexity of testing whether the result is empty? We show that even this problem is NP-hard.

Theorem 28: Let $\mathbf{Q}$ be a query on a database with schema $\mathbf{S}$ and instance $\mathbf{I}$. It is NP-hard to determine whether the result of $\mathbf{Q}$ on $\mathbf{I}$ is empty or not.

Proof: We reduce the problem to 3SAT [GJ79]. For the reduction we use the database and query schemas shown in Fig. 30, where the database schema is in the box on the right. We describe informally how to map an instance of 3SAT into a database instance and what the query $\mathbf{Q}$ is.

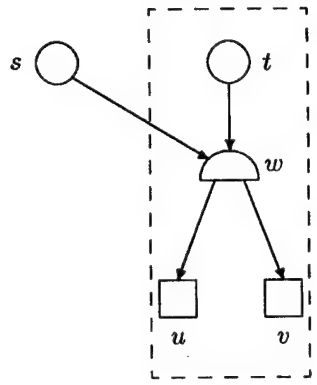


Figure 30: Reduction from 3SAT

The instance $\mathbf{I}$ corresponding to an instance of 3SAT is defined as follows. u contains all the variables in the instance and v contains the two constants T and F. w contains all possible pairs (x, T). Finally, each set in t corresponds to a clause, where the pair (x, T) is interpreted as the variable x and (x, F) as the variable $\overline{x}$.

Q is defined as follows. Each set in s corresponds to a satisfying truth assignment. In other words, such a set r must satisfy:

1. No two pairs (x, T) and (x, F) are in r. The pair (x, T) is now interpreted as assigning the value true to x.
2. For each x in u, either (x, T) or (x, F) is in r.
3. Each clause, i.e., each set $\{(x, a), (y, b), (z, c)\}$ in t is "satisfied" by the members of s, i.e., at least one of (x, a), (y, b) and (z, c) is in r.

It should be clear that we can write a formal LDM query expressing these requirements. It is then easily seen that instances for which 3SAT has a solution are mapped into database instances for which the query has a nonempty result and then testing whether the result is empty shows whether the instance of 3SAT has a satisfying truth assignment. ∎

A modification of the above proof shows that it is also NP-hard to determine whether a query **Q** is safe on a given database instance **I**. To see this, let the database be as in the proof of the above theorem, and let **Q** add the node s above, followed by a node q of type $\square$. The formula $\phi_s(x_s)$ is as before, while $\phi_q(x_q)$ will just require that the result at the node s be nonempty, e.g., by the formula $\phi_q(x_q) = (\exists x_s)(x_s =_l x_s)$. Note that ϕ_q does not mention the variable x_q.

If we map instances of 3SAT into database instances as above, then whenever the instance of 3SAT has a satisfying truth assignment, the result at s is nonempty. In that case ϕ_q is satisfied by any l-value, and the query is unsafe.

Conversely, whenever the instance of 3SAT has no satisfying truth assignment, the result at s is empty. But then ϕ_q is satisfied by no l-value and therefore **Q** is safe. This shows that a test for safety can be used to test satisfiability, and therefore that the problem of testing a query for safety on a given instance is NP-hard.

Chapter 7

The Algebraic Query Language

7.1. The Algebraic Operators

In this section we define a complete set of algebraic operators. We shall then show that any safe logical query is equivalent to some sequence of algebraic operations. Conversely, each algebraic operation is equivalent to a safe logical query.

Since a logical query adds some nodes to the database schema and leaves the instance of the database schema unchanged, each algebraic operator must do the same. So a selection operator, for example, should not delete tuples that do not satisfy the selection condition, but should rather create a copy of the database node. That copy should contain only those tuples that satisfy the condition. In fact this copying of tuples is really what is done in the relational model—a query does not throw away those tuples in the database that do not meet a selection condition, but rather copies those tuples that do. This issue is not addressed explicitly in relational database theory, since the theory does not deal with what happens to temporary relations that are created while computing the result of a query.

In this section **S** will be a database schema with instance **I**. The algebra will consist of operations of the form $w \leftarrow \alpha(v_1, \ldots, v_n)$. Here α is the name of the operator, and its arguments $v_1, \ldots, v_n$ are nodes in the schema **S**. α adds the node w to the schema, and extends **I** to the new schema. We define each operator as a simple logical query. To define each operator we give

1. The types of its arguments.
2. The type of w and the list of its children.
3. An LDM formula $\phi_w(x_w)$ that specifies the contents of $I(w)$.

7.1.1. Operators that Copy and Combine Existing Nodes

1. $w \leftarrow \square(v)$ creates a copy of the node v, as is shown in Fig. 31. In all these figures the schema **S** is shown in the box on the right, and the node that is created by the operation is on the left. For each distinct r-value in $I(v)$, $I(w)$ will contain exactly one l-value with this r-value. Note that duplication in $I(v)$ is eliminated in $I(w)$.
   (a) v is a node of **S** that has type $\square$.
   (b) w is of type $\square$.
   (c) $\phi_w(x_w)$ is $(\exists y_v)(x_w =_r y_v)$.

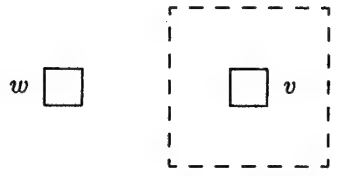


Figure 31: The algebraic operation $w \leftarrow \square(v)$

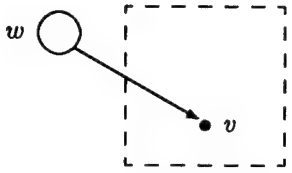


Figure 32: The algebraic operation $w \leftarrow \bigcirc(v)$

2. $w \leftarrow \square(d)$ creates a node of type $\square$ that contains just the constant d.
   (a) d is a constant in the data domain D.
   (b) w is of type $\square$.
   (c) $\phi_w(x_w)$ is $x_w =_r d$.
3. $w \leftarrow \bigcirc(v)$ creates a node that contains the powerset of $I(v)$ (see Fig. 32).
   (a) v is any node in the schema **S**.
   (b) w is of type $(\bigcirc, v)$.
   (c) $\phi_w(x_w)$ is T (i.e. always true).

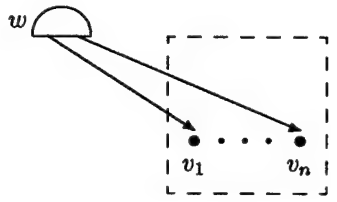


Figure 33: The algebraic operation $w \leftarrow \frown(v_1, \ldots, v_n)$

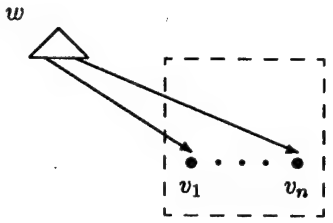


Figure 34: The algebraic operation $w \leftarrow \triangle(v_1, \ldots, v_n)$

4. $w \leftarrow \frown(v_1, \ldots, v_n)$ creates a node that contains the cartesian product $I(v_1) \times \cdots \times I(v_n)$ (see Fig. 33).
   (a) $v_1, \ldots, v_n$ are any n nodes in the schema **S**.
   (b) w is of type $(\frown, n, v_1, \ldots, v_n)$.
   (c) $\phi_w(x_w)$ is T.
5. $w \leftarrow \triangle(v_1, \ldots, v_n)$ creates a node that contains the disjoint union $I(v_1) \cup \cdots \cup I(v_n)$ (see Fig. 34).
   (a) $v_1, \ldots, v_n$ are n distinct nodes of the schema **S**.
   (b) w is of type $(\triangle\, n, v_1, \ldots, v_n)$.
   (c) $\phi_w(x_w)$ is T.

Example 18: In all the examples in this section **S** will be the genealogy whose schema is shown in Fig. 8 (on page 10). In most of the examples the instance will be that shown in Fig. 10 (page 11).

1. The operation $u' \leftarrow \square(u)$ adds the node u' to **S**, and extends the instance as shown in Fig. 35.

2. For the remainder of this example, the database instance will be the smaller instance in Fig. 36. The result of the operation $u' \leftarrow \bigcirc(u)$ is the schema shown in Fig. 37, together with the instance shown in Fig. 38.

3. The result of the operation $v' \leftarrow \frown(u, v)$ is the schema shown in Fig. 39, together with the instance in Fig. 40.

$I(u')$

l	$r(l)$
17	Rehoboam
18	Solomon
19	David
20	Batsheba
21	Jesse

Figure 35: Example of the algebraic operation $u' \leftarrow \square(u)$

$I(u)$

l	$r(l)$
1	Batsheba

$I(v)$

l	$r(l)$
2	(1, 3)

$I(w)$

l	$r(l)$
3	$\emptyset$

Figure 36: A smaller instance of the genealogy schema

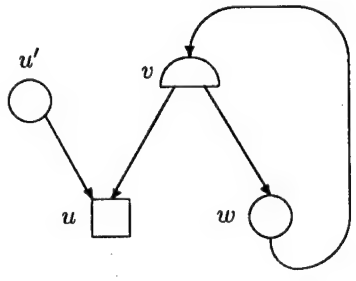


Figure 37: Example of the algebraic operation $u' \leftarrow \bigcirc(u)$

$I(u')$

l	$r(l)$
4	$\emptyset$
5	{Batsheba}

Figure 38: Result of $u' \leftarrow \bigcirc(u)$

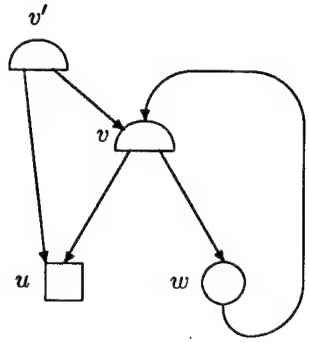


Figure 39: Example of the algebraic operation $v' \leftarrow \frown(u, v)$

$I(v')$

l	$r(l)$
4	(1, 2)

Figure 40: Result of the operation $v' \leftarrow \frown(u, v)$

7.1.2. Selection Operators

The LDM algebra has two selection operators.

1. The operation $w \leftarrow \sigma_{i\ \theta\ j}(v)$ is similar to the selection operation in the relational algebra. This operator selects those tuples in v whose i^{th} and j^{th} components are related by θ (see Fig. 41).

   (a) v is a node of **S** of type $(\frown, n, v_1, \ldots, v_n)$ and $i\ \theta\ j$ is one of the relations $i \in j$, $i\ \pi_t\ j$, $i\ \rho\ j$, $i =_l j$ and $i =_r j$.

   (b) w is of type $(\frown, n, v_1, \ldots, v_n)$.

   (c) $\phi_w(x_w)$ is

   $$(\exists x_v)(\exists y_{v_i})(\exists y_{v_j})(y_{v_i}\ \pi_{v_i}\ x_v \wedge y_{v_j}\ \pi_{v_j}\ x_v \wedge y_{v_i}\ \theta\ y_{v_j} \wedge x_v =_r x_w)$$

   Alternatively, θ may be of the form $i =_r d$ where d is a constant in D. Then $\phi_w(x_w)$ is

   $$(\exists x_v)(\exists y_{v_i})(y_{v_i}\ \pi_{v_i}\ x_v \wedge y_{v_i} =_r d \wedge x_v =_r x_w)$$

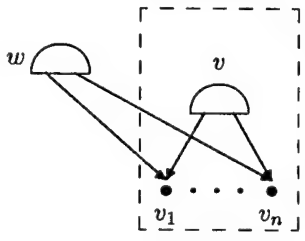


Figure 41: The algebraic operation $w \leftarrow \sigma_{i\ \theta\ j}(v)$

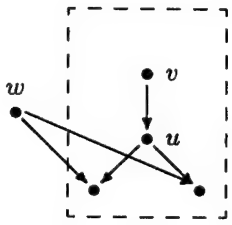


Figure 42: The algebraic operation $w \leftarrow \sigma_{\text{in}}(u, v)$

2. $w \leftarrow \sigma_{\text{in}}(u, v)$. Here u is a child of v, and w will contain those elements of $I(u)$ that actually appear in $I(v)$, i.e., depending on the type of v, those elements of $I(u)$ that occur either as members of sets, r-values or tuples (see Fig. 42).

   (a) u and v are nodes of **S** and u is a child of v.

(b) w is of the same type as u and has the same children.

(c) $\phi_w(x_w)$ depends on the type of v. Note that v cannot be of type $\square$ since it has a child u.

i. If v is of type $\frown$ with u as its i^{th} child then $\phi_w(x_w)$ is

$$(\exists x_u)(\exists x_v)(x_u =_r x_w \wedge x_u\ \pi_i\ x_v)$$

If there are multiple edges from v to u, we have to say which one we mean. In this case we shall use the notation $\sigma_{\text{in}}(u, v, i)$ to mean: use the i^{th} edge with tail v.

ii. If v is of type $\triangle$, then $\phi_w(x_w)$ is

$$(\exists x_u)(\exists x_v)(x_u =_r x_w \wedge x_u\ \rho\ x_v)$$

iii. If v is of type $\bigcirc$, then $\phi_w(x_w)$ is

$$(\exists x_u)(\exists x_v)(x_u =_r x_w \wedge x_u \in x_v)$$

Example 19: The schema continues to be that of Fig. 8 (page 10) and the instance is that of Table 10 (page 11).

1. The result of the operation $u' \leftarrow \sigma_{(1=_r\text{"Rehoboam"})}(v)$ is the schema shown in Fig. 43, together with the instance in Fig. 44.
2. The result of $u' \leftarrow \sigma_{\text{in}}(w, v)$ is the schema in Fig. 45, and the instance in Table 46. Note that in this example u' is simply a copy of w, since every set in $I(w)$ is a member of some tuple in $I(v)$

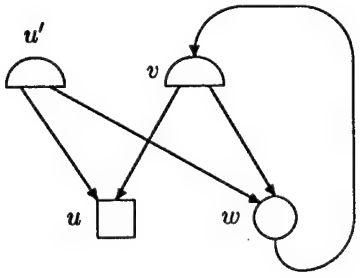


Figure 43: Example of selection

$I(u')$

l	$r(l)$
15	$(1, 11)$

Figure 44: Result of the operation $u' \leftarrow \sigma_{1=_r\text{"Rehoboam"}}$

7.1.3. Union, Difference and Projection

1. The union operator is similar to the relational union. The syntax we use is $w \leftarrow \cup(v_1, \ldots, v_n)$ (see Fig. 47).

(a) $v_1, \ldots, v_n$ are n nodes of **S** that are of the same type and have the same children.

(b) w has the same type and the same children as the v_i's.

(c) $\phi_w(x_w)$ is $(\exists x_{v_1})(x_{v_1} =_r x_w) \vee \cdots \vee (\exists x_{v_n})(x_{v_n} =_r x_w)$.

2. For difference we shall use infix notation, i.e., we shall write $w \leftarrow v_1 - v_2$ rather than $-(v_1, v_2)$.

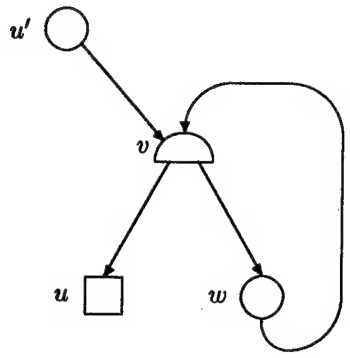


Figure 45: Example of the algebraic operation $u' \leftarrow \sigma_{\text{in}}(w, v)$

$I(u')$

l	$r(l)$
15	$\{7\}$
16	$\{8, 9\}$
17	$\{10\}$
18	$\emptyset$

Figure 46: Result of the algebraic operation $u' \leftarrow \sigma_{\text{in}}(w, v)$

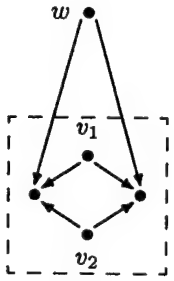


Figure 47: The algebraic operation $w \leftarrow \cup(v_1, v_2)$

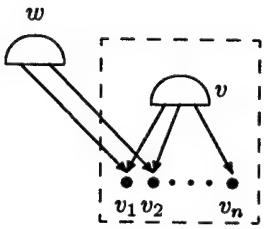


Figure 48: The algebraic operation $w \leftarrow \Pi_{\{v_1, v_2\}}(v)$

(a) v_1 and v_2 are nodes of **S** that are of the same type and have the same children.

(b) w has the same type and the same children as v_1 and v_2.

(c) $\phi_w(x_w)$ is $(\exists x_{v_1})(x_{v_1} =_r x_w) \wedge (\forall x_{v_2})(x_{v_2} \neq_r x_w)$.

3. The projection operation is similar to projection in the relational algebra. The syntax we use is $w \leftarrow \Pi_A(v)$, where A is an ordered multiset of edges with tail v.

(a) v is a node of **S** of type $(\frown, n, v_1, \ldots, v_n)$ and A is an ordered multiset of edges with tail v.

(b) Let $A = \{e_1, \ldots, e_k\}$ where e_j is the edge (v, v_{i_j}). Then w is of type $(\frown, k, v_{i_1}, \ldots, v_{i_k})$.

(c) $\phi_w(x_w)$ is

$$(\exists x_v)(\exists x_{v_1}) \cdots (\exists x_{v_n})\Big(x_{v_{i_1}}\ \pi_1\ x_w \wedge \cdots \wedge (x_{v_{i_k}}\ \pi_k\ x_w) \wedge x_v =_r (x_{v_1}, \ldots, x_{v_k})\Big)$$

When it will not cause any ambiguity, we shall use a set A of nodes rather than of edges, as in Fig. 48.

7.2. Equivalence of the Logical and Algebraic Query Languages

We can now use the algebraic operators defined in the previous section to define an algebraic query language. An algebraic query will be a sequence $\{\alpha_1, \ldots, \alpha_n\}$ of algebraic operators, where each α_i is an algebraic operator on the result of α_{i-1}. We want to show that this query language is equivalent to the logical query language. In other words, for each logical query on a schema **S**, there should exist a sequence of algebraic operations, and vice versa, with the property that the schemas created by these two queries are identical, and for every database instance **I**, the results are isomorphic relative to **S**. Unfortunately, as the next example shows, this is not quite true.

Example 20: Let **S** consist of a node u of type $\square$ and let **Q** be the logical query that adds a node v of type $\square$ to **S**. Let d_1 and d_2 be two distinct constants, and let $\phi_v(x_v)$ be $(x_v =_r d_1 \vee x_v =_r d_2)$. The candidate r-values for v are then $\{d_1, d_2\}$. There is no algebraic query equivalent to **Q**. If there was such a query, it would consist of one algebraic operation alone, since each operator adds a new node to the schema. By inspection we can see that no single algebraic operator is equivalent to **Q**.

How can we modify the definition to get an equivalent query? If $\mathbf{Q}_A$ is the algebraic query that consists of the operators $w_1 \leftarrow \square(d_1)$, $w_2 \leftarrow \square(d_2)$ and $v \leftarrow \cup(w_1, w_2)$ it is clear that the instance of v is what we are after. If we were then to restrict the result of the query to the schema that consists of the nodes u and v we get the instance we want. We have essentially used the two nodes w_1 and w_2 for temporary storage while computing the result of the query. In fact the same thing occurs in the relational model, since temporary relations are used there for subexpressions and then deleted at the end. It is therefore reasonable to expect the same thing to happen in the logical data model.

To be able to use temporary nodes, we extend the algebraic query language by adding a "delete" operator. This operator will delete a node from the schema and restrict the instance of the original schema to the new schema. We have to make sure that we never delete a node that is the child of some other node, since in that case the result would not be a legal schema. The operator that deletes the node v will be written $D(v)$.

Definition 30: Let **S** be an LDM schema with instance **I**. The algebraic operator $D(v)$ is legal when v is a node with no parent. The result of $D(v)$ is the schema $\mathbf{S}'$ that consists of deleting v from **S**, together with the instance that we get by restricting **I** to $\mathbf{S}'$.

In the algebraic query language we must take care not to delete database nodes, i.e., we must only allow the user to delete nodes that have been constructed by his query. We shall call the language with the deletion operator the *extended algebraic query language*.

Definition 31: Let **S** be an LDM schema. An *extended algebraic query* on **S** is a sequence $\mathbf{Q}_A = (\alpha_1, \ldots, \alpha_n)$ where each α_i is either

1. An operation of the form $w_i \leftarrow \beta_i(v_i^1, \ldots, v_i^{i_j})$, where β_i is an algebraic operator other than the deletion operator and $v_i^1, \ldots, v_i^{i_j}$ are either node of **S** or are nodes that were created by some previous β_j and have not been deleted.
2. The operator $D(v_i)$, where v_i is a node that was created by a previous algebraic operator in the sequence $\beta_1, \ldots, \beta_{i-1}$ and has not yet been deleted.

Definition 32: Let $\mathbf{Q}_A$ be an extended algebraic query on **S**, and let $\mathbf{Q}_B$ be an extended algebraic query on the result of $\mathbf{Q}_A$. The query $\mathbf{Q}_B \circ \mathbf{Q}_A$ is the *composition* of $\mathbf{Q}_A$ and $\mathbf{Q}_B$, formed simply by concatenating the lists of algebraic operators.

Obviously, the delete operator itself is not equivalent to any logical query, since every logical query adds nodes to the schema. This by itself does not necessarily mean that we cannot find a logical query equivalent to any extended algebraic query. After all, an extended algebraic query does not delete nodes of **S**, since the only nodes that are deleted are those that were constructed by previous algebraic operations. It might still be the case, as happened in Example 20, that there is an equivalent logical query that somehow expresses what the result of the query should be without using these temporary nodes.

At the end of this chapter (Theorem 36) we shall prove that such a query does not exist, thus showing that the extended algebraic query language is strictly more powerful than the logical query language. We can get equivalence by a simple modification of the logical query language: Allow logical queries to use temporary nodes as well.

Definition 33: Let **S** be an LDM schema. An *extended logical query* on **S** is a tuple $\mathbf{Q} = \left\langle S_{\mathbf{Q}}, \Phi_{\mathbf{Q}}, \prec_{\mathbf{Q}}, D_{\mathbf{Q}} \right\rangle$ where

1. $\left\langle S_{\mathbf{Q}}, \Phi_{\mathbf{Q}}, \prec_{\mathbf{Q}} \right\rangle$ is a logical query on **S**.
2. $D_{\mathbf{Q}}$ is the set of temporary nodes used in the query. $D_{\mathbf{Q}}$ is a subset of the query nodes $V_{\mathbf{Q}} - V$ that we can delete and still get an LDM schema. In other words, there is no edge with tail outside $D_{\mathbf{Q}}$ and head in $D_{\mathbf{Q}}$, i.e., if $(e_1, e_2) \in E_{\mathbf{Q}}$ and $e_2 \in D_{\mathbf{Q}}$ then $e_1 \in D_{\mathbf{Q}}$.

Definition 34: Let **Q** be the extended logical query $\mathbf{Q} = \left\langle S_{\mathbf{Q}}, \Phi_{\mathbf{Q}}, \prec_{\mathbf{Q}}, D_{\mathbf{Q}} \right\rangle$, and let **I** be an instance of **S**. The result of this query consists of

1. The schema $\mathbf{S}^*_{\mathbf{Q}}$ consisting of
   (a) The nodes in $V_{\mathbf{Q}} - D_{\mathbf{Q}}$.
   (b) The relevant edges, i.e., all those edges of $\mathbf{S}_{\mathbf{Q}}$ whose head and tail are both in $V_{\mathbf{Q}} - D_{\mathbf{Q}}$.
   (c) The restriction of the type assignment μ to $V_{\mathbf{Q}} - D_{\mathbf{Q}}$.
2. The result of **Q** on **I** is defined as follows. Let $\mathbf{I}_{\mathbf{Q}}$ be the result of $\left\langle \mathbf{S}_{\mathbf{Q}}, \Phi_{\mathbf{Q}}, \prec_{\mathbf{Q}} \right\rangle$ on **I**. The result of **Q** on **I** is then the restriction of $\mathbf{I}_{\mathbf{Q}}$ to $\mathbf{S}^*_{\mathbf{Q}}$.

We now start to prove the main result of this section, that the two extended query languages are equivalent. We start by proving that every extended algebraic query is equivalent to some extended logical query.

Lemma 29: Let $\mathbf{Q}_A = \{\alpha_1, \ldots, \alpha_n\}$ be an extended algebraic query on **S**. There exists a safe extended logical query $\mathbf{Q}_L$ on **S** such that for every instance **I** of **S**, the results of $\mathbf{Q}_A$ and $\mathbf{Q}_L$ on **I** are isomorphic relative to **S**.

Proof: The schema of $\mathbf{Q}_L$ will consist of all those nodes that are created by the operations in the query $\mathbf{Q}_A$. The set of temporary nodes $D_{\mathbf{Q}_L}$ will be the set of nodes deleted in $\mathbf{Q}_A$, i.e., $\{v_i \mid$ The operator α_i is $D(v_i)\}$. Since we are only allowed to delete nodes that are not in **S** and that have no parent, it is easy to see that there is no edge with tail outside $D_{\mathbf{Q}_L}$ and head in it. Each α_i that is not a delete operator must be of the form $w_j \leftarrow \beta_j(w_i^1, \ldots, w_i^{i_j})$. We define an order on the nodes of $V_{\mathbf{Q}} - V$ as follows: $w_i \prec w_j$ whenever $i < j$. $\phi_{w_i}(x_{w_i})$ is the formula that was used to define the operator β_j in the previous section. It is easy to verify that the results of $\mathbf{Q}_A$ and $\mathbf{Q}_L$ on any instance **I** are indeed isomorphic. ∎

We now show the converse. Let $\mathbf{Q}_L$ be a logical query on **S**. For the moment, we shall look at queries in the original, rather than the extended query language. Afterwards we shall see what to do with the extended

query language. Let **I** be a fixed instance of **S**. The definition of $\mathbf{Q}_A$ will not depend on **I**, but the results of $\mathbf{Q}_A$ and $\mathbf{Q}_L$ will only be isomorphic on those instances of **S** on which $\mathbf{Q}_L$ is safe. We keep **I** fixed just so we will be able to prove various lemmas about the results as we go along. Fig. 49 shows some of the nodes we construct in the algebraic query, and may help to understand the construction.

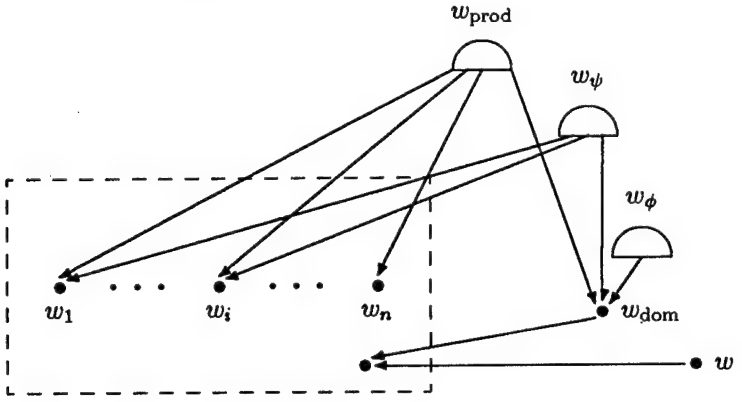


Figure 49: Constructing an equivalent algebraic query

We first look at the case when $\mathbf{Q}_L$ is a simple query $\mathbf{Q}_w$. We start by creating a node w_{dom}, that contains the "domain" of w, i.e., all those objects that might be candidate r-values for w if we were to ignore everything except the type of w and the fact that $\mathbf{Q}_L$ is safe on **I**. We define w_{dom} as follows.

1. If w is of type $\square$, let $v_1, \ldots, v_t$ be all the nodes in **S** that are of type $\square$ and let $d_1, \ldots, d_k$ be the constants that occur in $\phi_w(x_w)$. Define w_{dom} by the algebraic query:

$$
\begin{aligned}
& s_1 \leftarrow \square(v_1) \\
& \quad \vdots \\
& s_t \leftarrow \square(v_t) \\
& s_{t+1} \leftarrow \square(d_1) \\
& \quad \vdots \\
& s_{t+k} \leftarrow \square(d_k) \\
& w_{\text{dom}} \leftarrow \cup(s_1, \ldots, s_{t+k}) \\
& D(s_1) \\
& \quad \vdots \\
& D(s_{t+k})
\end{aligned}
$$

2. If $\mu(w) = (\backsim, k, v_1, \ldots, v_k)$ define w_{dom} by $w_{\text{dom}} \leftarrow \backsim(v_1, \ldots, v_k)$.
3. If $\mu(w) = (\mathrm{O}, v)$ define w_{dom} by $w_{\text{dom}} \leftarrow \mathrm{O}(v)$.
4. If $\mu(w) = (\triangle\!\!\!\!-\, k, v_1, \ldots, v_k)$ define w_{dom} by $w_{\text{dom}} \leftarrow \triangle\!\!\!\!-(v_1, \ldots, v_k)$.

We shall call this algebraic query $\mathbf{Q}_{\text{dom}}$. We formalize the intuition behind it in the following lemma.

Lemma 30:

1. The schema created by $\mathbf{Q}_{\text{dom}}$ is equal to the schema of $\mathbf{S}$ together with a node w_{dom} of the same type and with the same children as the node w in the original logical query $\mathbf{Q}_w$.
2. Let $\mathbf{I}_{\text{dom}}$ be the result of $\mathbf{Q}_{\text{dom}}$ on $\mathbf{I}$ and let $\mathbf{I}_w$ be the result of $\mathbf{Q}_w$ on $\mathbf{I}$. If r is an r-value in $I_w(w)$, then r is also an r-value in $\mathbf{I}_{\text{dom}}(w_{\text{dom}})$.

Proof: If w is of type $\frown$, $\triangle$or $\bigcirc$, the lemma is obvious. If w is of type $\square$, the first part follows from the fact that all the nodes except w_{dom} that are created by $\mathbf{Q}_{\text{dom}}$ are also deleted by it. The second part is an immediate consequence of Lemma 24 (page 43) and the definition of $\mathbf{Q}_{\text{dom}}$. ∎

We may assume, if necessary by renaming some bound variables, that all the bound variables in the formula $\phi_w(x_w)$ that was used to define $\mathbf{Q}_w$ are distinct. Let these variables be $x^1_{w_1}, \ldots, x^k_{w_k}$. The algebraic query $\mathbf{Q}_{\text{prod}}$ on the result of $\mathbf{Q}_{\text{dom}}$ consists of the algebraic operation

$$w_{\text{prod}} \leftarrow \frown(w_1, \ldots, w_k, w_{\text{dom}})$$

For the purpose of defining $\mathbf{Q}_A$ we are going to label the edges with head w_{prod} as follows. The i^{th} edge with head w_{prod} will be labeled $x^i_{w_i}$. These labels will be used only to define the algebraic query, and are not themselves part of the query.

In certain cases, when we create a new node using some algebraic operation, the outgoing edges from the new node will inherit the labels of the corresponding edges whose head is one of the arguments of the operator. We shall only use this inheritance in cases when it is unambiguous, i.e., in cases when all the arguments have the same labeling. The operations for which labels will be inherited are $\sigma_{i\ \theta\ j}$, difference and union. When we use the projection operation the new edges will also inherit the labeling of the corresponding edges whose head is the argument of the projection. These labels are essentially used to remind us which bound variable the edge corresponds to.

Arrange all the well formed subformulas of $\phi_w(x_w)$ in a list $\psi_1, \ldots, \psi_m$, where $\psi_m = \phi_w(x_w)$ and ψ_i precedes ψ_j whenever it is a subformula of ψ_j. For each such subformula, we shall define an extended algebraic query $\mathbf{Q}_{\psi_i}$ on the result of $\mathbf{Q}_{\psi_{i-1}}$. $\mathbf{Q}_{\psi_1}$ will be a query on the result of $\mathbf{Q}_{\text{prod}}$. The labels on the edges with tail w_ψ will correspond to the variables that might be free in ψ—i.e., those that haven't yet been bound by ψ. The node w_{ψ_i} will be of type $(\frown, j, w_{j_1}, \ldots, w_{j_k}, w_{\text{dom}})$, and will contain, intuitively, those tuples $(l_1, \ldots, l_k, l_d)$ for which $\models_{\mathbf{I}_{\psi_i}} \psi_i(l_1, \ldots, l_k, l_d)$.

1. ψ_i is $x^a_{w_a}\ \theta\ x^b_{w_b}$. $\mathbf{Q}_{\psi_i}$ consists of the algebraic operation $w_{\psi_i} \leftarrow \sigma_{a\ \theta\ b}(w_{\text{prod}})$.
2. ψ_i is $x^a_{w_a}\ \theta\ x_w$. $\mathbf{Q}_{\psi_i}$ consists of the algebraic operation $w_{\psi_i} \leftarrow \sigma_{a\ \theta\ k+1}(w_{\text{prod}})$
3. ψ_i is $x_w\ \theta\ x_w$. $\mathbf{Q}_{\psi_i}$ consists of the algebraic operation $w_{\psi_i} \leftarrow \sigma_{k+1\ \theta\ k+1}(w_{\text{prod}})$
4. ψ_i is $x^a_{w_a} =_r d$. $\mathbf{Q}_{\psi_i}$ consists of the algebraic operation $w_{\psi_i} \leftarrow \sigma_{a=_r d}(w_{\text{prod}})$.
5. ψ_i is $x_w =_r d$. $\mathbf{Q}_{\psi_i}$ consists of the algebraic operation $w_{\psi_i} \leftarrow \sigma_{(k+1)=_r d}(w_{\text{prod}})$
6. ψ_i is $\psi_{j_1} \vee \psi_{j_2}$. Let A_1 be the (ordered multi)set of edges with tail $w_{\psi_{j_1}}$ that have the same label as some edge with tail $w_{\psi_{j_2}}$. Let A_2 be the corresponding set of edges with tail $w_{\psi_{j_2}}$. $\mathbf{Q}_{\psi_i}$ is the following extended algebraic query

$$\begin{aligned} s_1 &\leftarrow \Pi_{A_1}(w_{\psi_{j_1}}) \\ s_2 &\leftarrow \Pi_{A_2}(w_{\psi_{j_2}}) \\ w_{\psi_i} &\leftarrow \cup(s_1, s_2) \\ &D(s_1) \\ &D(s_2) \end{aligned}$$

(s_1 and s_2 are different temporary nodes from those used above, and from similarly named nodes used below.) Note that the way we defined A_1 and A_2 guarantees that there is no ambiguity in labeling the edges of the result, at least as long as the labels of the edges in A_1 and A_2 are in the same order. We shall show later that this is indeed the case.

7. ψ_i is $\neg\psi_j$. Let A be the (ordered multi)set of edges with tail w_{dom} that have the same label as some edge with tail w_{ψ_j}. $\mathbf{Q}_{\psi_i}$ is the following extended algebraic query

$$\begin{aligned} s_1 &\leftarrow \Pi_A(w_{\psi_{\text{dom}}}) \\ w_{\psi_i} &\leftarrow s_1 - w_{\psi_j} \\ D(s_1) & \end{aligned}$$

As in the previous case we shall be able to label the edges with tail w_{ψ_i} without any ambiguity.

8. ψ_i is $(\exists x^a_{w_a})(\psi_j)$. Let A be the (ordered multi)set of all edges with tail w_{ψ_j} except for the edge labeled $x^a_{w_a}$. We shall show later that there must be exactly one edge with such a label. $\mathbf{Q}_{\psi_i}$ then consists of the algebraic operation $w_{\psi_i} \leftarrow \Pi_A(w_{\psi_j})$.

Lemma 31: Let $\psi = \psi_i$ be one of these well formed subformulas of $\phi_w(x_w)$. Let $x^{a_1}_{w_{a_1}}, \ldots, x^{a_j}_{w_{a_j}}$ be those variables in the above list that are not bound in ψ_i. Note that some of the $x^{a_i}_{w_{a_i}}$'s may not actually occur in ψ_i. Then w_{ψ_i} is of type $(\frown, (a_j + 1), w_{a_1}, \ldots, w_{a_j}, w_{\text{dom}})$, and the t^{th} edge with tail w_{ψ_i} has head w_{a_t} and is labeled with the variable $x^{a_t}_{w_{a_t}}$. As a consequence of this, all the labelings of edges are in the same order and the assumptions that we made on the labelings when we defined the w_{ψ_i}'s hold.

Proof: The proof is a fairly straightforward induction using the definition of w_{ψ_i}. The tricky case is when ψ_i is $\psi_{j_1} \vee \psi_{j_2}$. Then the children of $w_{\psi_{j_1}}$ correspond to the bound variables of ϕ_w that are not bound in ψ_{j_1} and the children of $w_{\psi_{j_2}}$ to the bound variables of ϕ_w not bound in ψ_{j_2}. Since a variable is not bound in $\psi_{j_1} \vee \psi_{j_2}$ iff it is not bound in ψ_{j_1} and it is not bound in ψ_{j_2}, we see that the result does hold in this case. ∎

Lemma 32: Let w_{ψ_i} be of type $(\frown, j, w_{j_1}, \ldots, w_{j_k}, w_{\text{dom}})$. Let $\mathbf{I}_{\psi_i}$ be the result of $\mathbf{Q}_{\psi_i}$ on $\mathbf{I}$, let l_d be a member of $I_{\psi_i}(w_{\text{dom}})$ and let l_t be a member of $I_{\psi_i}(w_{j_t})$ for $t = 1, \ldots, k$. Then there exists an l in $I_{\psi_i}(w_{\psi_i})$ with $r(l) = (l_1, \ldots, l_k, l_d)$ if and only if $\models_{\mathbf{I}_{\psi_i}} \psi_i(l_1, \ldots, l_k, l_d)$. Intuitively, $(l_1, \ldots, l_k, l_d)$ is a "candidate r-value" iff it satisfies ψ_i.

Proof: A straightforward induction on the structure of ψ_i. ∎

The extended algebraic query $\mathbf{Q}_{\text{final}}$ on the result of $\mathbf{Q}\phi_w$ consists of the following operations

$$\begin{aligned} w_A &\leftarrow \sigma_{\text{in}}(w_{\text{dom}}, w_\phi) \\ D(w_\phi) & \\ &\vdots \\ D(w_{\psi_1}) & \\ D(w_{\text{prod}}) & \\ D(w_{\text{dom}}) & \end{aligned}$$

We finally define the algebraic query $\mathbf{Q}_A$ as

$$\mathbf{Q}_{\text{final}} \circ \mathbf{Q}_\phi \circ \mathbf{Q}_{\psi_{m-1}} \circ \mathbf{Q}_{\psi_1} \circ \mathbf{Q}_{\text{prod}} \circ \mathbf{Q}_{\text{dom}}$$

Lemma 33: Let $\mathbf{I}_1$ be the result of $\mathbf{Q}_w$ on $\mathbf{I}$ and let $\mathbf{I}_2$ the result of the algebraic query $\mathbf{Q}_A$ on $\mathbf{I}$. Then $\mathbf{I}_1$ and $\mathbf{I}_2$ are isomorphic relative to $\mathbf{S}$.

Proof: First note that the schemas are equal. The only node created but not deleted by $\mathbf{Q}_A$ is the node w_A. This node is similar to the node w_{dom} and hence to w.

We have to show that the instances of w_A and w are isomorphic, i.e., that at the point in evaluating the queries that we compute the instances of these nodes, they have the same candidate r-values. We assume that we are at the point in the evaluation of $\mathbf{Q}_A$ just before the final round of deletions.

Let r be a candidate r-value for w. Extend $\mathbf{I}$ to an instance $\mathbf{I}_w$ of $\mathbf{S}_{\mathbf{Q}_w}$ by defining $I_w(w) = \{l\}$ and $r(l) = r$. Then $\models_{\mathbf{I}_w} \phi_w(l)$. Let $\mathbf{I}_{\phi_w}$ be the result of $\mathbf{Q}_{\phi_w}$ on $\mathbf{I}$. By Lemma 30 part 2, r is a candidate r-value for w_{dom} and so for some l_d in $I_{\phi_w}(w_{\text{dom}})$, $r(l_d) = r$. By Lemma 18 (page 41), $\models_{\mathbf{I}_w} \phi_w(l)$ implies $\models_{\mathbf{I}_{\phi_w}} \phi_w(l_d)$, and therefore, by Lemma 32, for some l_ϕ in $I_{\phi_w}(w_{\text{dom}})$, $r(l_\phi) = r$ is a candidate r-value for w_A.

For the converse, suppose that r is a candidate r-value for w_A. Let $\mathbf{I}_{\phi_w}$ be the result of $\mathbf{Q}_{\phi_w}$ on $\mathbf{I}$. Since r is a candidate r-value for w_A, for some l_ϕ in $I_{\phi_w}(w_{\phi_w})$ and some l_d in $I_{\phi_w}(w_{\text{dom}})$, $r(l_\phi) = (l_d)$ and $r(l_d) = r$. Since l_ϕ is in $I_{\phi_w}(w_{\phi_w})$, Lemma 30 implies that $\models_{\mathbf{I}_{\phi_w}} \phi_w(l_d)$. Restrict $\mathbf{I}_{\phi_w}$ to an instance $\mathbf{I}_{\text{dom}}$ of the schema of $\mathbf{Q}_{\text{dom}}$. Then $\models_{\mathbf{I}_{\text{dom}}} \phi_w(l_d)$, and so by Lemma 18, $r(l_d) = r$ is a candidate r-value for w. ∎

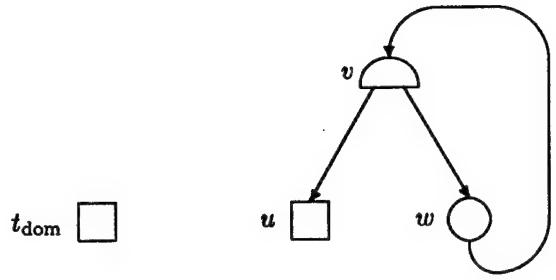
We can easily extend this to general queries by concatenating the algebraic queries for the individual simple queries. If we have an extended logical query we have to add deletion operations at the end of the algebraic query that delete those nodes in the delete set of the query. This completes the proof of the following theorem.

Theorem 34: The extended algebraic query language and the extended logical query language are equivalent, i.e., for every extended algebraic query on $\mathbf{S}$ there exists a safe extended logical query on $\mathbf{S}$ and for every extended logical query on $\mathbf{S}$ there exists an extended algebraic query on $\mathbf{S}$, such that both queries define the same schema and for every database instance $\mathbf{I}$ on which the logical query is safe, the results of both queries are isomorphic relative to $\mathbf{S}$. ∎

Example 21: We shall illustrate the proof of Theorem 34 by showing how it would construct an extended algebraic query equivalent to the query $\mathbf{Q}_1$ in Example 12 (page 37). We shall name the new node in that query t rather than u'. The database instance is shown in Table 10 (page 11).

1. $\mathbf{Q}_{\text{dom}}$ consists of the algebraic operations $s_1 \leftarrow \square(u)$, $t_{\text{dom}} \leftarrow \cup(s_1)$ followed by $D(s_1)$, the deletion of s_1. Note that the union of copies of all database nodes of type $\square$ becomes here the union of a single node. This is of course superfluous, but as we are illustrating the proof of the theorem, rather than showing how to compute the result efficiently, we include this operation. The final schema (after the deletion) and the instances of the nodes are shown in Fig. 50.
2. $\mathbf{Q}_{\text{prod}}$ consists of the operation $t_{\text{prod}} \leftarrow \bigcirc(u, t_{\text{dom}})$. Fig. 51 shows the schema after this operation. The instance is too large to show here. It contains 25 l-values, 25–49, with all the possible pairs in $\{1, \ldots, 5\} \times \{20, \ldots, 24\}$ as r-values.
3. The subformulas of ϕ are $\psi_1 = (x_t =_r y_u)$ and $\phi = \psi_2 = (\exists y_u)(x_t =_r y_u)$. $\mathbf{Q}_{\psi_1}$ is $t_{\psi_1} \leftarrow \sigma_{2=_r 1}(w_{\text{prod}})$, and its result is shown in Fig. 52.
4. $\mathbf{Q}_\phi$ is $t_\phi \leftarrow \Pi_{\{t_{\text{dom}}\}}(t_{\psi_1})$. Its result is shown in Fig. 53.
5. $\mathbf{Q}_{\text{final}}$ consists of the operation $t_A \leftarrow \sigma_{\text{in}}(t_{\text{dom}}, t_\phi)$ followed by the deletion of all the temporary nodes. The result of this is shown in Fig. 54.

This example shows that the algebraic query that we get from the proof, as is also the case in Codd's proof of the equivalence of the relational algebra and tuple calculus, is not necessarily the best way to actually evaluate a logical query. Our example could be done much more efficiently by the single algebraic operation $t_A \leftarrow \square(u)$.



$I(s_1)$

l	$r(l)$
15	Rehoboam
16	Solomon
17	David
18	Batsheba
19	Jesse

$I(t_{\text{dom}})$

l	$r(l)$
20	Rehoboam
21	Solomon
22	David
23	Batsheba
24	Jesse

Figure 50: Result of $\mathbf{Q}_{\text{dom}}$

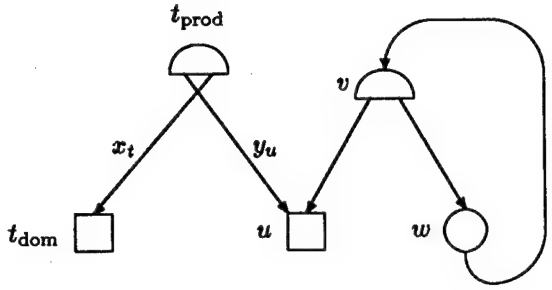


Figure 51: Schema of $\mathbf{Q}_{\text{prod}}$

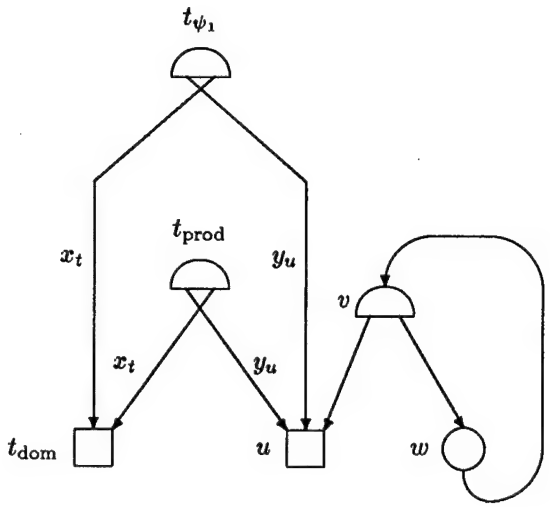
7.3. Various Results about the Algebra

Most of the algebraic operators are natural analogues of relational operators. Even the powerset operator $\bigcirc(v)$ is fairly natural, since it creates the entire domain of the node, and is therefore similar to the operator $\frown(v_1, \ldots, v_n)$ that is based on the cross product. The exception is the restriction operator, $\sigma_{\text{in}}(u, v)$. Even though it is a type of selection, there is an essential difference between it and the other LDM selection operator. Restriction selects objects based on whether they are used in some other node, whereas the other selection operator selects objects based only on some property of the object by itself. For this reason, the LDM selection operator resembles the relational selection while restriction does not.

For this reason, it would be nice if we were able to eliminate the restriction operator from the algebra, i.e., to show that it can be expressed in terms of the other algebraic operators. We now show that this is impossible.

Theorem 35: The extended algebraic query language is strictly more powerful than the language without restriction.

Proof: Let the database schema consist of the nodes u and v in Fig. 55. We claim that there is no extended algebraic query not using restriction that is equivalent to the query $w \leftarrow \sigma_{\text{in}}(u, v)$. To see why this is true, note that the only algebraic operators apart from restriction that can create nodes of type $\square$ are the operators



$I(t_{\psi_1})$

l	$r(l)$
50	(1,20)
51	(2,21)
52	(3,22)
53	(4,23)
54	(5,24)

Figure 52: Result of $\mathbf{Q}_{\psi_1}$

1. $t \leftarrow \square(v_1)$
2. $t \leftarrow \square(d)$
3. $t \leftarrow \cup(v_1, \ldots, v_n)$
4. $t \leftarrow v_1 - v_2$

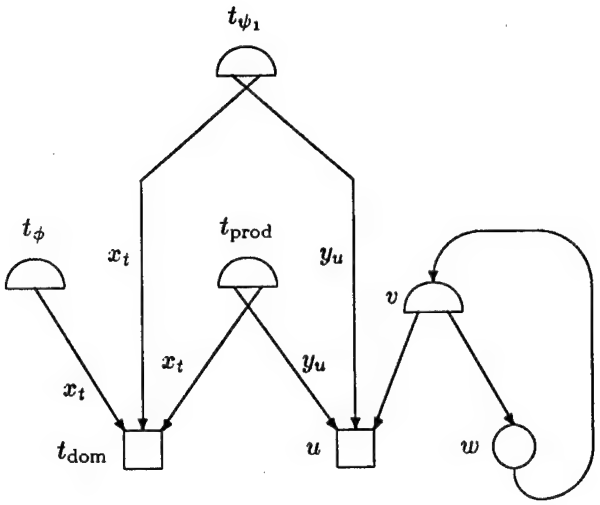
and in all these cases the arguments must also be of type $\square$. Intuitively, the query must therefore construct the node w without looking at the node v at all. More formally, suppose that there existed an extended algebraic query $\mathbf{Q}$ equivalent to $w \leftarrow \sigma_{\text{in}}(u, v)$. Let $\mathbf{I}_1$ and $\mathbf{I}_2$ be the following database instances. Both $I_1(u)$ and $I_2(u)$ are equal to $\{1, 2\}$, where $r_1(1) = r_2(1) = a$ and $r_1(2) = r_2(2) = b$. On the other hand, $I_1(v) = I_2(v) = \{3\}$ but $r_1(3) = (1)$ and $r_2(3) = (2)$. The candidate r-values for w on these instances should be, respectively, a and b. We shall show, by induction on the length of $\mathbf{Q}$, that for any node t in $\mathbf{Q}$ of type $\square$, in particular w, the candidate r-values for t on both $\mathbf{I}_1$ and $\mathbf{I}_2$ are the same. For $\mathbf{Q}$ of length 0, the result is obvious. Assume that the inductive hypothesis holds for all queries of length less than n, and let $\mathbf{Q}$ be of length n. If the last operation in $\mathbf{Q}$ is a deletion or if the last operation creates a node of type other than $\square$, the result is immediate. If the last operation creates a node of type $\square$, it must do so using one of the operations 1–4 above, and then the result follows immediately from the inductive assumption. ∎

Our second result shows that the extended algebraic query language is strictly more powerful than the nonextended logical query language.

Theorem 36: There is an extended algebraic query that is not equivalent to any (nonextended) logical query.

Proof: We shall show that there is an extended logical query not equivalent to any (nonextended) logical query. The result will then follow by Theorem 34.

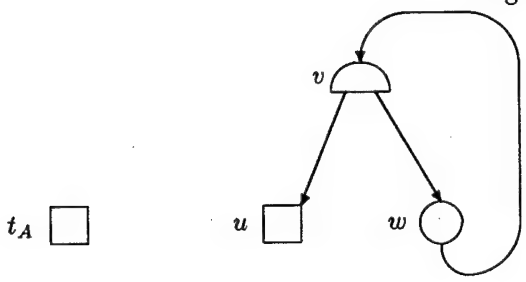
The database schema $\mathbf{S}$ will be the same as the one we used in the proof of Theorem 26 (page 44). The extended query $\mathbf{Q}$ will also be the same as in that proof, together with the set of temporary nodes $\{u, w\}$.



$I(t_\phi)$

l	$r(l)$
55	(20)
56	(21)
57	(22)
58	(23)
59	(24)

Figure 53: Result of $\mathbf{Q}_\phi$



$I(t_A)$

l	$r(l)$
60	Rehoboam
61	Solomon
62	David
63	Batsheba
64	Jesse

Figure 54: Result of $\mathbf{Q}_{\text{final}}$

An equivalent logical query would have to define the contents of t in terms of the contents of v alone, which we showed in the proof of Theorem 26 to be impossible. ∎

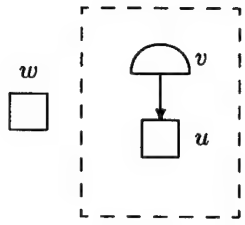


Figure 55: Proof that restriction is essential

Chapter 8

Elimination of Cycles

8.1. Introduction

LDM schemas can contain cycles not only in the schemas but also in the data. For example, if $l \in r(l)$ for some l-value l, then we would have a cycle in the data. Having introduced cycles into the model, we would like to study their expressive power. The problem we shall look at is: Are there applications that cannot be modeled without cycles? For example, consider the following schema.

Example 22: Fig. 56 shows an example of a cyclic database schema that stores information about procedure calls in a program. The schema is the same as the genealogy schema that we have used up to now. Elements in $I(v)$ represent procedures, elements in $I(u)$ represent procedure names and elements in $I(w)$ represent sets of procedures. Thus, if $x \in I(v)$ and $r(x) = (y, z)$, then $r(y)$ is the name of the procedure x and $r(z)$ is the set of procedures called from x. Note that if a procedure calls itself, then we have a cycle in the data. This is the reason we do not use the genealogy example, since the data in the genealogy should not be cyclic. An acyclic schema that intuitively seems to "capture the same information" is shown in Fig. 57. In this schema, elements in $I(v_2)$ represent procedure entities, elements in $I(u)$ represent procedure names, elements in $I(w)$ represent sets of procedures, and elements in $I(v_1)$ represent the relationship "procedure calls procedures."

To formalize the idea of "capturing the same information," we use a definition, closely related to the notion of query-equivalence of [Hul84]. Intuitively, two schemas capture the same information if we can map instances of one schema to instances of the other, and queries on one schema to queries on the other, such that the result of the first query on the first instance is isomorphic to the result of the second query on the second instance.

In our query language, however, the result of a query is not necessarily a new, independent schema, but may contain pointers to nodes in the database schema **S**. Because of this, it is meaningless to talk in general about isomorphism between the results of the queries. In order for such an isomorphism to be meaningful we shall restrict the query language, in this chapter, to a language that does not allow pointers to the database.

Definition 35: A *independent query* on a schema **S** consists of a new schema $\mathbf{S_Q}$ together with an ordering of nodes and a set of LDM formulas, such that when we add the query schema to the database schema we get an LDM query.

The result of an independent query is defined in the obvious way. If two independent queries on different database schemas have the same query schema, we are then able to talk about their results being isomorphic.

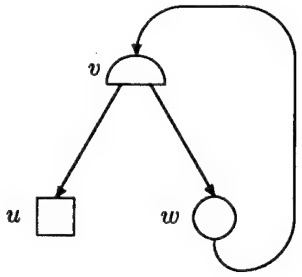


Figure 56: Cyclic schema

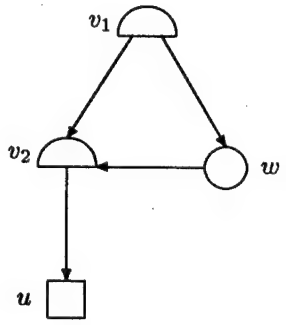


Figure 57: An acyclic schema equivalent to it

Definition 36: Let **S** and **T** be schemas. Then **T** *dominates* **S** if there is a mapping f of instances of **S** to instances of **T** such that for each independent query $\mathbf{Q}_1$ on **S**, there is an independent query $\mathbf{Q}_2$ on **T** such that $\mathbf{Q}_1(\mathbf{I})$ is isomorphic to $\mathbf{Q}_2(f(\mathbf{I}))$ for all instances **I** of **S** on which $\mathbf{Q}_1$ is safe. We say that **S** and **T** are *equivalent* if each of them dominates the other.

We shall only be able to prove that the two schemas in Example 22 are equivalent when the relationship represented by v_1 is functional, i.e., when for each procedure is related to exactly one set of procedures, i.e. those procedures that it calls. This means that in fact we do not have an equivalence between the schemas, but rather between the original schema and a new, constrained one. To make our results more general, we shall also start off with a constrained schema, so that we shall in fact show an equivalence between a constrained schema $(\mathbf{S}, \phi)$ and an acyclic constrained schema $(\mathbf{T}, \psi)$. We start by describing the general transformation from cyclic schemas to equivalent acyclic schemas. The idea is to break cycles by creating composition nodes that represent the cyclic relationships, as in the above example.

8.2. Converting Cyclic Schemas to Acyclic Ones

When we try to break cycles in arbitrary cyclic schemas, we notice that there are several pathological cases in which the above method does not work. First of all, the cycle has to contain a ⬭ or △-node at which to break it, i.e., it cannot consist just of ○-nodes. The method also fails when the cycle contains such a node of type ⬭ or △ but this node has only one child. If we break the cycle at such a node, we would end up with a childless ⬭ or △-node after breaking the cycle. In both of these cases, the schema relates l-values to l-values without relating them at any point to the actual data. Intuitively, pure relationships between l-values such as these do not correspond to anything in the "real world," which justifies looking only at schemas without such a relationship.

We make one further restriction on the LDM schemas. If a cycle in the schema contains a node of type △, our method of removing cycles appears not to work. For example, if the cyclic schema was the one shown in Fig. 58, it would be converted into the schema in Fig. 59, that does not represent the same structure. The original schema essentially stores data objects at the top node, along with sets of objects, sets of sets of

objects, etc, and this is not what is stored in the acyclic schema. For this reason, we shall require that nodes of type △occur only outside cycles. Unlike the other conditions, this is a real restriction on the power of our method, and more work remains to be done on whether cycles involving nodes of type △can be eliminated.

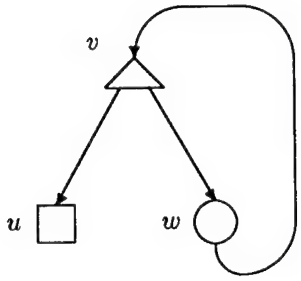


Figure 58: A cyclic schema

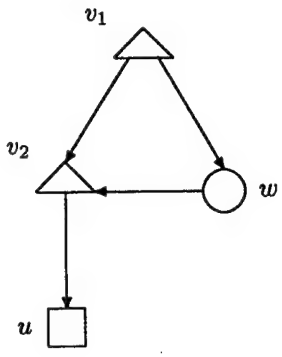


Figure 59: Corresponding acyclic schema

Definition 37: A schema **S** is called *well-formed* if from each node in the schema there is a path to a node of type □, and no node of type △occurs in any cycle of **S**.

Definition 38: Let **S** be a cyclic schema. A node v in **S** is called a *possible breakpoint* if

1. It is of type ⌓.
2. It is in at least one cycle.
3. It has at least one child that is not in any cycle.

For example, the node v in Fig. 60 is a possible breakpoint.

Lemma 37: Let **S** be a well-formed schema. Either **S** is cyclic, or it has a possible breakpoint.

Proof: Assume that **S** contains a cycle, and let u_0 be a node in that cycle. Since **S** is well-formed, there is a path $u_0, \ldots, u_n$ from u_0 to a node u_n of type □. Let u_k be the first node on this path that is not in any cycle; there must be at least one such node, since the node u_n is not in any cycle. We claim that the node u_{k-1} is a possible breakpoint. By definition, it is on at least one cycle and has a child that is not in any cycle. Since u_{k-1} is in a cycle and that cycle does not contain u_k, u_{k-1} must have at least one other child. Since **S** is well-formed, u_{k-1} must be of type ⌓. ∎

Let **S** be a well-formed cyclic schema, and let v be a possible breakpoint. There are two ways to generalize Example 22. One way is to break one cycle through v at a time. In some cases this can result in unnecessarily complicated schemas, and we prefer instead to break the cycles that go through v all at once. The node v is replaced by two nodes v_1 and v_2 (see Fig. 61). All the edges that had head v, except for those that belonged to one of the cycles through v, will now have head v_1. All the edges that had tail v, except for those that were in the cycles, now have tail v_2. v_1 and v_2 will both be of type ⌓. The formal definition is as follows.

Definition 39: Let $(\mathbf{S}, \phi)$ be a well-formed cyclic constrained schema, where $\mathbf{S} = \langle V, E, \mu \rangle$, and let v be a possible breakpoint. Then $\mathrm{Br}(\mathbf{S}, \phi, v)$ is the constrained schema $(\mathbf{S}', \psi)$ where $\mathbf{S}' = \langle V', E', \mu' \rangle$ and $\psi = \mathrm{Br}(\phi, v)$ are defined as follows.

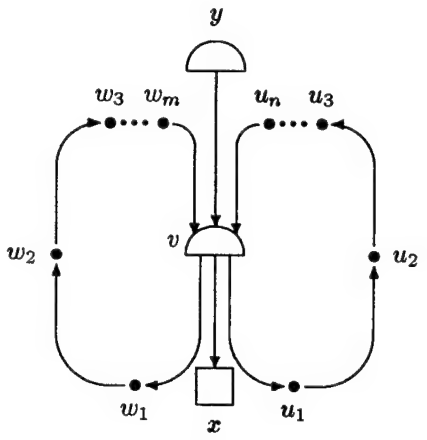


Figure 60: Cycles through v

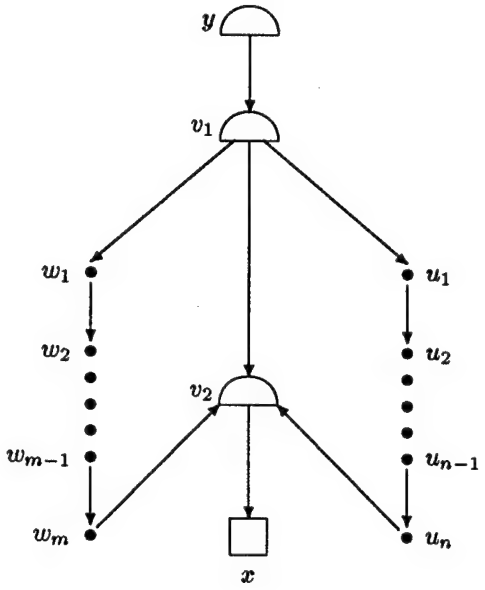


Figure 61: After breaking the cycles

1. We introduce two new nodes v_1 and v_2. V' has v replaced by v_1 and v_2, i.e., $V' = V - \{v\} \cup \{v_1, v_2\}$.
2. All the nodes in V' except for v_1 and v_2 have the same type as in **S**, i.e., $\mu'(u) = \mu(u)$ for all u in $V - \{v\}$. v_1 and v_2 are both of type ⌓.
3. Let C be the set of nodes that are on the cycles that go through v. Then E' is defined as follows

$$\begin{aligned} E' = E \quad & - \quad \{(u_1, u_2) \mid (u_1, u_2) \in E, u_1 = v \text{ or } u_2 = v\} \\ & \cup \quad \{(v_1, v_2)\} \\ & \cup \quad \{(u, v_1) \mid (u, v) \in E, u \notin C\} \\ & \cup \quad \{(u, v_2) \mid (u, v) \in E, u \in C\} \\ & \cup \quad \{(v_2, u) \mid (v, u) \in E, u \notin C\} \\ & \cup \quad \{(v_1, u) \mid (v, u) \in E, u \in C\} \end{aligned}$$

In other words, the edges in the new schema are

(a) All those in the original schema, except for those whose head or tail is v.

(b) v_2 is a child of v_1.

(c) Each edge with head v is replaced by

i. An edge with head v_1, when the edge is not part of a cycle.

ii. An edge with head v_2, when the edge is part of a cycle.

(d) Each edge with tail v is replaced by

i. An edge with tail v_1, when the edge is part of a cycle.

ii. An edge with tail v_2, when the edge is not part of a cycle.

Any edge that replaces an edge with head or tail v has the same position in the ordering as the edge it replaces. The edge from v_1 to v_2 follows all the other edges in the ordering.

In order to define the new constraint $\psi = \mathrm{Br}(\phi, v)$, we first define a function $F(\phi)$ from LDM formulas over $\mathbf{S}$ to LDM formulas over $\mathbf{S}'$.

Definition 40: The function $F(\phi)$ is defined by induction on the size of ϕ as follows.

1. (a) If u and w are not equal to v, then $F(x_u\ \pi_u\ y_w)$ is $x_u\ \pi_u\ y_w$.
   (b) If w is not equal to v, then
      i. If w is not on a cycle through v, then $F(x_v\ \pi_v\ y_w)$ is $x_{v_1}\ \pi_v\ y_w$.
      ii. If w is on a cycle through v, then $F(x_v\ \pi_v\ y_w)$ is $x_{v_2}\ \pi_v\ y_w$.
   (c) If u is not equal to v, then
      i. If u is not on a cycle through v, then $F(x_u\ \pi_u\ y_v)$ is $x_u\ \pi_u\ y_{v_2}$.
      ii. If w is on a cycle through v, then $F(x_u\ \pi_u\ y_v)$ is $x_u\ \pi_u\ y_{v_1}$.
   (d) $F(x_v\ \pi_t\ y_v)$ is $x_{v_2}\ \pi_{v_2}\ y_{v_1}$.
2. $F(x_v \in y_w)$ is defined similarly, except that we do not have to worry about the cases when $w = v$.
3. $F(x_u\ \rho\ y_w)$ is similar to the previous case.
4. (a) If u is not equal to v, then $F(x_u =_l y_u)$ is $x_u =_l y_u$.
   (b) $F(x_v =_l y_v)$ is $x_{v_1} =_l y_{v_1}$.
5. $F(x_u =_r d)$ is $x_u =_r d$.
6. $F(\phi \wedge \psi)$ is $F(\phi) \wedge F(\psi)$.
7. (a) If u is not equal to v, then $F((\forall x_u)\phi)$ is $(\forall x_u)F(\phi)$.
   (b) $F((\forall x_v)\phi)$ is $(\forall x_{v_1})(\forall x_{v_2})((x_{v_2}\ \pi_{v_2}\ x_{v_1}) \wedge F(\phi))$, where x_{v_1} and x_{v_2} are new variables, and the projection uses the last edge from v_1 to v_2.

Definition 41: The constraint ϕ is mapped into the constraint

$$\mathrm{Br}(\phi, v) = F(\phi) \wedge (\forall x^a_{v_1})(\forall x^b_{v_1})(\forall x_{v_2})(x_{v_2}\ \pi_{v_2}\ x^a_{v_1} \wedge x_{v_2}\ \pi_{v_2}\ x^b_{v_1} \Rightarrow x^a_{v_1} =_l x^b_{v_1}) \wedge (\forall x_{v_2})(\exists x_{v_1})(x_{v_2}\ \pi_{v_2}\ x_{v_1})$$

The last two conjuncts express the functional relationship that exactly one v_2 is associated with each v_1.

Lemma 38: Let $(\mathbf{S}, \phi)$ be a well-formed cyclic schema, and let v be a possible breakpoint. Then $\mathrm{Br}(\mathbf{S}, \phi, v) = (\mathbf{S}', \psi)$ is also well-formed.

Proof: We first have to show that $\mathbf{S}'$ is a legal LDM schema. The only reason it may fail to be one is that $\mathbf{S}'$ may contain a node of type $\bigcirc$ that has no children. It is clear from the definition of $\mathbf{S}'$ that the only node where this could happen is v_2, but v_2 has at least one child since v has at least one child that is not in any cycle through v.

We now show that $\mathbf{S}'$ is well formed. Let w be an arbitrary node of $\mathbf{S}'$. We have to show that there is a path in $\mathbf{S}'$ from w to a node of type $\square$, and that no cycle contains a node of type $\triangle$ To prove the second of these, it is not hard to show that we can convert a cycle in $\mathbf{S}'$ into a cycle in $\mathbf{S}$ by replacing all occurrences of v_1 and v_2 in the cycle by the node v. For the first condition there are two cases.

1. w is neither of the nodes v_1 and v_2. Then w must be a node of $\mathbf{S}$. Since $\mathbf{S}$ is well-formed, there must be a path in $\mathbf{S}$ from w to a node x of type $\square$. Let $w, w_1, \ldots, w_n, x$ be a shortest such path. Clearly x is also in $\mathbf{S}'$. If all the other nodes on the path are also in $\mathbf{S}'$, we are done. Otherwise, one of these nodes, say w_i, is equal to v, and by the minimality of the path there is at most one such w_i. Since v is a possible breakpoint, it has a child u_1 that is not on any cycle, and therefore there is a path $u_1, \ldots, u_m$ in $\mathbf{S}$ from u_1 to a node u_m of type $\square$ (see Fig. 62). There are then two possibilities.

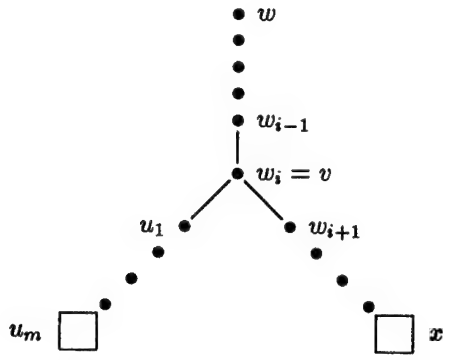


Figure 62: Proof of Lemma 38

(a) w_{i-1} is not on any cycle through v. Then $w, \ldots, w_{i-1}, v_1, v_2, u_1, \ldots, u_m$ is a path in $\mathbf{S}'$ from w to a node u_m of type $\square$.

(b) w_{i-1} is on a cycle through v. Then $w, \ldots, w_{i-1}, v_2, u_1, \ldots, u_m$ is a path in $\mathbf{S}'$ from w to a node u_m of type $\square$.

2. w is either v_1 or v_2. If we have a path from v_2 to a node x of type $\square$, we can easily convert it to a path from v_1 to x by using the edge (v_1, v_2). Assume therefore that w is the node v_2. Since v is a possible breakpoint, it has a child w_1 that is not in any cycle. Since $\mathbf{S}$ is well-formed, there is a path from w_1 to a node x of type $\square$. Let $w_1, \ldots, w_n, x$ be the shortest such path in $\mathbf{S}$. Then no node on this path is equal to v and therefore $v_2, w_1, \ldots, w_n, x$ is a path in $\mathbf{S}'$ to a node x of type $\square$. ∎

We now show that if we repeatedly break cycles, we eventually get an acyclic schema.

Lemma 39: Let $(\mathbf{S}, \phi)$ be a well-formed constrained cyclic schema. If we repeatedly break cycles in $\mathbf{S}$ at possible breakpoints, we shall eventually get an acyclic constrained schema $(\mathbf{T}, \psi)$. The termination does not depend on the order in which we choose the breakpoints.

Proof: The proof is by induction on the number of nodes of the schema that are in at least one cycle. We show that whenever we break a cycle we reduce the number of such nodes by at least one. Let $(\mathbf{S}_1, \phi_1)$ be the schema before breaking the cycles through v and let $(\mathbf{S}_2, \phi_2) = \mathrm{Br}(\mathbf{S}_1, \phi_1, v)$ be the schema afterwards. We show that

1. The two new nodes v_1 and v_2 are not in any cycle in $\mathbf{S}_2$.

2. Any node in $\mathbf{S}_1$ other than v that is not in any cycle in $\mathbf{S}_1$, is also not in any cycle of $\mathbf{S}_2$.

Together, these conditions immediately imply the result.

1. Assume that there is a cycle in $\mathbf{S}_2$ that goes through v_1 or through v_2. Let C be the shortest such cycle. There are three cases

   (a) C contains v_1 but not v_2. Let w be the node in C that immediately precedes v_1. By replacing v_1 in C by v we get a cycle in $\mathbf{S}_1$. But then, when we construct $\mathbf{S}_2$, we replace the edge (w, v) by the edge (w, v_2), and $\mathbf{S}_2$ then contains no edge from w to v_1.

(b) C contains v_2 but not v_1. Let u be the node in C that immediately succeeds v_2. In a similar way, by replacing v_2 by v, we get a cycle in $\mathbf{S}_1$. Therefore, when constructing $\mathbf{S}_2$, we replace the edge (v, u) by the edge (v_1, u), and $\mathbf{S}_2$ does not contain an edge from v_2 to u.

(c) C contains both v_1 and v_2. v_2 must occur immediately after v_1 on C, since otherwise we could shorten the cycle C by replacing the path from v_1 to v_2 by the edge (v_1, v_2). Let u be the node in C that immediately precedes v_1. If we replace v_1 and v_2 in C by the node v, we get a cycle in $\mathbf{S}_1$. But then the edge (u, v_1) would be replaced in $\mathbf{S}_2$ by the edge (u, v_2) and $\mathbf{S}_2$ would not contain an edge from u to v_2, a contradiction.

2. Let u be a node of $\mathbf{S}_1$ that does not appear in any cycle, and assume that u is in some cycle C in $\mathbf{S}_2$. As we have just shown, no cycle in $\mathbf{S}_2$, and in particular C, can contain either of the nodes v_1 or v_2. But then C is also a cycle in $\mathbf{S}_1$, a contradiction. ∎

8.3. Equivalence of the Schemas

We first show how to map an instance of $(\mathbf{S}, \phi)$ into an instance $\mathrm{Br}(\mathbf{I}, v)$ of $(T, \psi) = \mathrm{Br}(\mathbf{S}, \phi, v)$. The intuition behind the construction is as follows. We got from $\mathbf{S}$ to $\mathbf{T}$ by breaking cycles through v. The instance of any node other than v, that does not have v as either a parent or a child, is not changed. Each l-value in $I(v)$ is replaced by a pair of l-values, one in $\mathrm{Br}(\mathbf{I}, v)(v_1)$ and the other in $\mathrm{Br}(\mathbf{I}, v)(v_2)$. The second of these l-values is the child of the first. We then modify the r-values of the l-values in the parents and children of v in a straightforward way.

Definition 42: Let $(\mathbf{S}, \phi)$ be a well-formed, cyclic, constrained schema, and let $\mathbf{I}_1 = (I_1, r_1)$ be an instance of it. Let v be a possible breakpoint, and let $(\mathbf{T}, \psi) = \mathrm{Br}(\mathbf{S}, \phi, v)$. Then $\mathbf{I}_2 = (I_2, r_2) = \mathrm{Br}(\mathbf{I}_1, v)$ is the instance of $\mathbf{T}$ that is defined as follows. For each l in $I_1(v)$, we introduce two new l-values, that will be written as $\alpha(l)$ and $\beta(l)$.

1. $I_2(v_1)$ is defined as $\{\alpha(l) \mid l \in I_1(v)\}$ and $I_2(v_2)$ as $\{\beta(l) \mid l \in I_1(v)\}$, i.e., they contain all these new l-values. Since v is of type $\frown$, for each such l in $I_1(v)$, $r_1(l) = (l_1, \ldots, l_n)$ for some $l_1, \ldots, l_n$. Assume, w.l.o.g., that the first i children of v are those that are in cycles through v. Then $r_2(\alpha(l))$ is defined as $(l_1, \ldots, l_i, \beta(l))$ and $r_2(\beta(l))$ as $(l_{i+1}, \ldots, l_n)$. If the j^{th} child of v is v itself, then the corresponding component of $r_2(\alpha(l))$ will be $\beta(l_j)$ instead of l_j.

2. If w is any node except v that is not a parent of v, then $I_2(w) = I_1(w)$, and for each l in this set, $r_2(l) = r_1(l)$.

3. If w is a node (except v) that is a parent of v, then $I_2(w)$ is defined as $I_1(w)$. For the r-values, there are two cases to consider.

   (a) w is not in any cycle that goes through v. If w is of type $(\frown, n)$, then for each l in $I_1(w)$, $r_1(l) = (l_1, \ldots, l_n)$ for suitable l_i's. Let v be the i^{th} child of w Then $r_2(l)$ is defined as $(l_1, \ldots, l_{i-1}, \alpha(l_i), l_{i+1}, \ldots, l_n)$. This generalizes easily to the case when there multiple edges from w to v. The other possibility is that w is of type $(\bigcirc, v)$. In that case, for each l in $I_2(w)$, $r_2(l)$ is defined as $\{\alpha(l') \mid l' \in r_1(l)\}$.

   (b) w is on a cycle through v. The r-values are defined as in the previous case, but with $\beta(l)$ used everywhere instead of $\alpha(l)$.

Lemma 40: Let $\mathbf{I}$ be an instance of $(\mathbf{S}, \phi)$ and let v be a possible breakpoint. Let $\mathbf{I}^*$ be the instance $\mathrm{Br}(I, v)$. Then $\mathbf{I}^*$ is an instance of the schema $(\mathbf{T}, \psi) = \mathrm{Br}(\mathbf{S}, \phi, v)$.

Proof: We have to show that $\mathbf{I}^*$ satisfies the constraint $\mathrm{Br}(\phi, v)$. It is clear that it satisfies the two final conjuncts in the definition of $\mathrm{Br}(\phi, v)$, since there is a 1–1 correspondence between l-values $\alpha(l)$ in $\mathbf{I}^*(v_1)$ and l-values $\beta(l)$ in $\mathbf{I}^*(v_2)$. It remains to show that $\models_{\mathbf{I}^*} F(\phi)$. This is a consequence of the following assertion, whose proof is a routine induction on the structure of the formula ϕ.

Let $\psi(x^1_{w_1}, \ldots, x^n_{w_n}, y^1_v, \ldots, y^m_v)$ be an arbitrary LDM formula over $\mathbf{S}$ where all the variables of sort v are at the end. Then the free variables of $F(\psi)$ are $x^1_{w_1}, \ldots, x^n_{w_n}$, $y^1_{v_1}, y^1_{v_2}, \ldots, y^m_{v_1}, y^m_{v_2}$. If $l_i \in I(w_i)$ for all i, $1 \leq i \leq n$, and $l'_i \in I(v)$ for $i = 1, \ldots, m$ then

$$\models_{\mathbf{I}} \psi(l_1, \ldots, l_n, l'_1, \ldots, l'_m) \Leftrightarrow \models_{\mathbf{I}^*} F(\psi)(l_1, \ldots, l_n, \alpha(l'_1), \beta(l'_1), \ldots, \alpha(l'_m), \beta(l'_m)) \blacksquare$$

Lemma 41: Let $(\mathbf{S}, \phi)$ be a well-formed constrained cyclic schema, let v be a possible breakpoint, and let $(\mathbf{T}, \psi)$ be the schema that we get by breaking the cycles that go through v. Then $(\mathbf{T}, \psi)$ dominates $(\mathbf{S}, \phi)$.

Proof: Let $\mathbf{Q}_1$ be an independent query on $\mathbf{S}$. $\mathbf{Q}_2$ will consist of the same schema as $\mathbf{Q}_1$ and its nodes will be in the same order. Each formula $\phi_w(x_w)$ in $\mathbf{Q}_1$ is replaced by the corresponding formula $F(\phi_w)(x_w)$ in $\mathbf{Q}_2$. It is clear that we get a logical query.

Let $\mathbf{I}$ be a fixed instance of $(\mathbf{S}, \phi)$ and let $\mathbf{I}^*$ be the instance $\mathrm{Br}(\mathbf{I}, v)$ of $(\mathbf{T}, \psi)$. We show that the result $\mathbf{I}_1$ of $\mathbf{Q}_1$ on $\mathbf{I}$ and the result $\mathbf{I}^*_1$ of $\mathbf{Q}_2$ on $\mathbf{I}^*$ are isomorphic. The isomorphism is defined using the topological order on the query nodes, as follows.

Assume that we have defined the isomorphism f between $\mathbf{I}_1$ and $\mathbf{I}^*_1$ on all the query nodes that precede the node u. For each $l \in I_1(u)$, let $r = r_1(l)$ be its r-value. We define an r-value r' as follows

1. If v is of type $\frown$, then r is a tuple $(l_1, \ldots, l_n)$. We define r' to be the tuple $(f(l_1), \ldots f(l_n))$.
2. If v is of type $\bigcirc$, then r is a set, and we define r' to be the set $\{f(\tilde{l}) \mid \tilde{l} \in r\}$.
3. If v is of type $\triangle$ then $r = \tilde{l}$, and we define r' to be $f(\tilde{l})$.
4. If v is of type $\square$, then $r \in D$, and we define r' to be equal to r.

It is straightforward to show that r' is a candidate r-value for u in $\mathbf{Q}_2$. This gives us a 1–1 correspondence between the r-values of $I_1(u)$ and those of $I^*_1(u)$. If we then define $f(l) = l^*$, where l^* is the l-value in $I^*_1(u)$ with r-value r', we extend the isomorphism f to u. By repeating this for each query node u, we get an isomorphism between $\mathbf{I}_1$ and $\mathbf{I}^*_1$. $\blacksquare$

We now define the inverse mapping on instances.

Definition 43: Let $(\mathbf{S}, \phi)$ and $(\mathbf{T}, \psi)$ be as above, and let $\mathbf{I}_2$ be an instance of $(\mathbf{T}, \psi)$. Then $\mathbf{I}_1$ is the following instance

1. $I_1(v)$ is defined as $I_2(v_1)$. Whenever $l \in I_1(v)$, $r_1(l)$ will be a tuple containing all the components from $r_2(l)$, except for the last one that corresponds to the new edge to v_2, together with all the components of $r_2(\Pi_{v_2}(l))$.
2. If w is any node except v_1 and v_2 and w is not a parent of v_2, then $I_1(w) = I_2(w)$ and the r-values are the same as in $\mathbf{I}_2$.
3. If w is any node except v_1 that is not a parent of v_2 then $I_1(w) = I_2(w)$. If w is of type $(\frown, n)$ with v_2 as its k^{th} child, then each l in $I_2(w)$ has an r-value of the form $r_2(l) = (l_1, \ldots, l_k, \ldots, l_n)$ for suitable l_i's. Since $\mathbf{I}_2$ satisfies ψ, there is a unique l^*_k in $I_2(v_1)$ with l_k as its last component. We then define $r_1(l) = (l_1, \ldots, l^*_k, \ldots, l_n)$. We define the r-values for nodes of type $\bigcirc$ in a similar way.

Lemma 42: Let $(\mathbf{S}, \phi)$, $(\mathbf{T}, \psi)$, $\mathbf{I}_2$ and $\mathbf{I}_1$ be as in the above definition. Then $\mathbf{I}_1$ is an instance of $(\mathbf{S}, \phi)$.

Proof: $\mathbf{I}_1$ clearly is an instance of $\mathbf{S}$. The proof that $\models_{\mathbf{I}_1} \phi$ is a straightforward induction on the structure of ϕ, similar to the proof of Lemma 40. ∎

It is easy to show that the two mappings on instances are inverses of each other, i.e., applying one and then the other yields an instance isomorphic to the original one. To complete the proof of equivalence we show that $(\mathbf{S}, \phi)$ dominates $(\mathbf{T}, \psi)$.

Lemma 43: Let $(\mathbf{S}, \phi)$ be a well-formed constrained cyclic schema, let v be a possible breakpoint and let $(\mathbf{T}, \psi)$ be the schema that we get by breaking the cycles that go through v. Then $(\mathbf{S}, \phi)$ dominates $(\mathbf{T}, \psi)$.

Proof: Let $\mathbf{Q}_2$ be an independent query on $\mathbf{S}$. $\mathbf{Q}_1$ consists of the same schema and node ordering as $\mathbf{Q}_2$. The formula $\phi_w(x_w)$ in $\mathbf{Q}_2$ is replaced in $\mathbf{Q}_1$ by the following formula. Each variable in ϕ_w of the form x_{v_1} or x_{v_2} is replaced by a variable x_v. These variables are distinct, i.e., x_{v_1} and x_{v_2} are replaced by *different* variables. The only other change we have to make in ϕ_w is to atomic formulas that involve π_k. Formulas of the form $y_w\ \pi_j\ x_{v_1}$ and $y_w\ \pi_j\ x_{v_2}$ are replaced by $y_w\ \pi_k\ x_v$, where w is the k^{th} child of v. The remaining possibility, $x_{v_1}\ \pi_{n+1}\ y_{v_2}$, where v has n children, is replaced by $x_v =_l y_v$. Proving the equivalence of $\mathbf{Q}_1$ and $\mathbf{Q}_2$ is now straightforward, making use of the fact that we only consider instances that satisfy the constraint ψ. ∎

Combining Lemmas 41 and 43, we get

Lemma 44: Let $(\mathbf{S}, \phi)$ be a well-formed constrained cyclic schema, let v be a possible breakpoint and let $(\mathbf{T}, \psi)$ be the schema we get by breaking the cycles that go through v. Then $(\mathbf{S}, \phi)$ and $(\mathbf{T}, \psi)$ are equivalent.

Finally, by applying this result repeatedly together with Lemma 39, we get the desired result.

Theorem 45: Let $(\mathbf{S}, \phi)$ be a well-formed constrained schema. There exists an acyclic constrained schema $(\mathbf{T}, \psi)$ that is equivalent to $(\mathbf{S}, \phi)$. ∎

Chapter 9

Conclusions

We have described a new model of data, the Logical Data Model, that is designed to combine the advantages of the existing data models. On the one hand, it enables the database to describe more of the semantics of the data than is possible using the relational model of data. On the other hand, we do not lose the nice properties that relational databases have, in particular the ability to query the database using equivalent non-procedural and procedural languages.

Some directions for future work are as follows.

1. More work has to be done on the query language. The languages we have defined are similar to the initial versions of Codd's relational algebra and tuple calculus. We have outlined in Section 3.2 how the LDM languages could be modified to obtain a more user-friendly and efficient language, but more work has to be done in this direction before an implementation would be possible.

2. Another direction for future work is extending the power of the query language. While there does not appear to be any need for the full power of the implicitly defined queries in our first attempt, there may be specific constraints that we want the result of the query to satisfy, and these may not be expressible by node-by-node formulas. Appendix B gives one example of the sort of difficulties we encounter in one extension of this sort.

3. The query languages that we have described are all first-order. Recent papers, such as [HN84] [Rei78] [Ull85], have proposed using a more powerful query language, similar to PROLOG, for accessing databases. Such a language would be able, among other things to compute the transitive closure of a relation, something that cannot be done in the relational algebra [AU79]. It may be possible to extend the LDM query language along these lines to get a non first-order language without the problems that arise with Jacobs' database logic.

4. More work remains to be done on the expressive power of cyclicity. It is still open whether cycles containing nodes of type $\triangle$ can be eliminated. Furthermore, we have only shown that, according to a *certain* measure, cycles in well-formed schemas do not add any expressive power. But it is not clear that this measure is the ultimate one.

Appendix A

An Early Attempt to Define the Query Language

A.1. Introduction

In this appendix, $\mathbf{S}$ will be a fixed database schema, $\mathbf{I}$ an instance of it and $\mathbf{Q} = \left\langle \mathbf{S_Q}, \prec_\mathbf{Q}, \phi_\mathbf{Q} \right\rangle$ a query on $\mathbf{S}$.

One of our attempts to extend the relational model to LDM schemas was the following. A query consists of an extension of $\mathbf{S}$ together with some sentence that specifies an instance of it. In other words

Definition 44: A query $\mathbf{Q}$ on $\mathbf{S}$ consists of

1. An extension $\mathbf{S_Q}$ of $\mathbf{S}$.
2. An LDM sentence $\phi_\mathbf{Q}$ over $\mathbf{S_Q}$.

The result of the query should be an extension of $\mathbf{I}$ that satisfies sentence $\phi_\mathbf{Q}$.

Definition 45: The result of $\mathbf{Q}$ on $\mathbf{I}$ is an extension $\mathbf{I_Q}$ of $\mathbf{I}$ to $\mathbf{S_Q}$ such that $\models_{\mathbf{I_Q}} \phi_\mathbf{Q}$.

One problem with this definition is that there may be many different ways to extend $\mathbf{I}$, all of which satisfy $\phi_\mathbf{Q}$. One way we tried to deal with this problem was to require that a query have a unique result, i.e., put the burden on the user to make sure that he only asks such queries. Uniqueness, of course, will only be up to isomorphism. In the relational model the term *safe queries* is used to denote queries for which the result is defined. The only thing that could go wrong there is that the result may be infinite, and that is in fact the definition of safety in the relational model. For LDM queries this is no longer true, since there are other things that may be wrong with a query. For example, there may be no extension of $\mathbf{I}$ to $\mathbf{S_Q}$ that satisfies the query, or there may be several possible such extensions. We shall borrow the term *safe query* from the relational model and use it with an extended meaning. It will denote those queries that have a unique result. Note that since we are only interested in finite instances the safe queries in the relational model turn out to be a special case of our more general definition.

Definition 46: A query is *safe up to isomorphism* if for every instance $\mathbf{I}$ of $\mathbf{S}$ there is a unique extension of $\mathbf{I}$ to $\mathbf{S_Q}$ that satisfies $\phi_\mathbf{Q}$. The uniqueness is up to isomorphism relative to $\mathbf{S}$.

The problem with this definition is that requiring that a query be safe up to isomorphism is too strong a requirement.

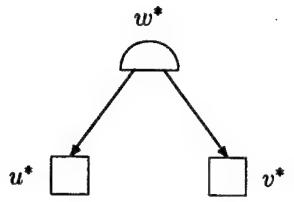


Figure 63: A logical query

Example 23: Let the database schema be the genealogy schema **S** shown in Fig. 8 (page 10) Suppose that as a query on **S** we want to construct the LDM schema that corresponds to the relational model. In other words, the query schema is the extension of **S** formed by adding the nodes in Figure 63 to **S**. We want w^* to contain pairs of l-values that correspond to (Person-Parent) pairs. To get these pairs, look at elements of $I(v)$. For each such element, take its left component and pair it with those elements that we get by taking the elements in its right component, and finding the person that they point to. When we write this out formally, we get the following LDM sentence

$$\begin{aligned}\phi_1 = \quad & (\forall x^1_{w^*})(\forall x^2_{u^*})(\forall x^3_{v^*})\Big((x^2_{u^*}\pi_{u^*}x^1_{w^*}) \wedge (x^3_{v^*}\pi_{v^*}x^1_{w^*}) \\ & \Rightarrow (\exists y^1_u)(\exists y^2_u)(\exists y^3_v)(\exists y^4_w)(\exists y^5_v)\Big((y^1_u =_r x^2_{u^*}) \wedge \Big(y^2_u =_r x^3_{v^*}\Big) \wedge \Big(y^3_v =_r (y^1_u, y^4_w)\Big) \\ & \wedge \Big(y^5_v \in y^4_w\Big) \wedge \Big(y^2_u \pi_u y^5_v\Big)\Big)\Big)\end{aligned}$$

We can think of ϕ_1 as being similar to how we would express a query in the relational model. In other words, we say what the objects in the result should satisfy. The problem with this query is that it is not safe up to isomorphism. One reason for this is that unlike the relational model the LDM model can express duplication of data. Everything in the result of the above query does indeed correspond to a (Person-Parent) pair but there is nothing to stop such a pair from appearing twice or more often. To prevent this from happening, we have to add another sentence to the query. The following sentence, ϕ_2, says explicitly that the result contains no duplication.

$$\begin{aligned}\phi_2 \quad = \quad & (\forall x_{u^*})(\forall y_{u^*})(x_{u^*} \neq_l y_{u^*} \Rightarrow x_{u^*} \neq_r y_{u^*}) \\ \wedge \quad & (\forall x_{v^*})(\forall y_{v^*})(x_{v^*} \neq_l y_{v^*} \Rightarrow x_{v^*} \neq_r y_{v^*}) \\ \wedge \quad & (\forall x_{w^*})(\forall y_{w^*})(x_{w^*} \neq_l y_{w^*} \Rightarrow x_{w^*} \neq_r y_{w^*})\end{aligned}$$

The query with $\phi_1 \wedge \phi_2$ is still unsafe. Nothing in what we have written so far says that any particular (Person-Parent) pair must appear in the result—we have only said that everything in the result is such a pair and that nothing appears more than once. In the relational model this is something that we do not have to say explicitly—we just say what should be in the result and the result will then contain one copy of each tuple that satisfies the query. To make our query safe we can add another sentence, ϕ_3, to the query. ϕ_3 says explicitly that anything in **I** that corresponds to a (Person-Parent) pair gives rise to a corresponding

tuple in the result.

$$\phi_3 = (\forall y_v^1)(\forall y_u^2)(\forall y_w^3)(\forall y_v^4)(\forall y_u^5)\Big((y_u^2 \pi_u y_v^1) \wedge (y_w^3 \pi_w y_v^1) \wedge (y_v^4 \in y_w^3) \wedge (y_u^5 \pi_u y_v^4)$$
$$\Rightarrow (\exists x_{w^*}^1)(\exists x_{u^*}^2)(\exists x_{v^*}^3)\Big(\Big(x_{w^*}^1 =_r (x_{u^*}^2, x_{v^*}^3)\Big) \wedge \Big(x_{u^*}^2 =_r y_u^2\Big) \wedge \Big(x_{v^*}^3 =_r y_u^5\Big)\Big)\Big)$$

Finally, to get a safe query, we have to restrict the contents of $I(u^*)$ and $I(v^*)$, i.e, to say that these nodes contain nothing that is not needed for the tuples in $I(w^*)$. Formally

$$\phi_4 = (\forall x_{u^*})(\exists y_{w^*})(x_{u^*} \pi_{u^*} y_{w^*}) \wedge (\forall x_{v^*})(\exists y_{w^*})(x_{v^*} \pi_{v^*} y_{w^*})$$

Putting all this together we get the query $\mathbf{Q} = (S_\mathbf{Q}, \phi_\mathbf{Q})$ where

$$\phi_\mathbf{Q} = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4$$

While this query is safe at last, it is obviously far too complicated to be any real use.

A.2. Safety up to Duplication

What the above example shows is that to get a safe query we have to add conjuncts to our original query that say things that are obvious. One of these explicitly states the fact that there is no duplication in the result. We could simplify what the user has to write by making this part of the definition of the query language, which could be done by having the query processor automatically add a conjunct similar to ϕ_2 to the query. We feel that this is not the right way to proceed for several reasons. One reason is that this seems a rather ad hoc approach. Why add this conjunct rather than other ones? The other reason is that the query may, either implicitly or explicitly, require that there be some duplication in the result. If the system were then to add ϕ_2 to the query it would convert what was originally a safe query into an unsafe one.

The alternative way to proceed that seems preferable from a mathematical viewpoint as well is to keep the original query as the condition that the result must satisfy but also require that the result have as little duplication as possible. In our example this would mean that there is no duplication at all but in general that would not have to be the case. A safe query would then be one that has a unique *minimal instance* satisfying the query. In such a case we shall say the query is *safe up to duplication*.

Essentially, an instance is minimal if there is no smaller instance that satisfies the sentence. Some difficulty occurs when trying to define what minimality means at a node of type O. If v is of type (O, u) and the query requires that u have some duplication, we have to make sure that we minimize internal duplication in the sets, i.e., that we take only one copy among the duplicates in u as a member of each set in v (unless duplication of this sort is also required by the query).

In order to define minimality, we first define a relation $l_1 \preceq l_2$ on l-values. $l_1 \preceq l_2$ will mean, intuitively, that while l_1 and l_2 contain the same information, l_1 may contain more internal duplication than l_2.

Definition 47: Let $\mathbf{I}_1 = \langle I_1, r_1 \rangle$ and $\mathbf{I}_2 = \langle I_2, r_2 \rangle$ be two extensions of $\mathbf{I}$ to $\mathbf{S}_\mathbf{Q}$. We say that an element l_1 of $I_1(v)$ is *dominated* by an element l_2 of $I_2(v)$, and write $l_1 \preceq l_2$, iff

1. If v is a node of the database schema $\mathbf{S}$, then $l_1 = l_2$. This means that different l-values in the database are regarded as essentially different objects, even if their r-values are the same.
2. If v is a query node, i.e., v is a node in $V_\mathbf{Q} - V$, then

(a) If $\mu(v) = \square$, then $r_1(l_1) = r_2(l_2)$.

(b) If $\mu(v) = (\backsim, n)$, then $\Pi_i(l_1) \preceq \Pi_i(l_2)$ for all i, $1 \leq i \leq n$.

(c) If $\mu(v) = \triangle$ then $r_1(l_1) \preceq r_2(l_2)$.

(d) If $\mu(v) = \bigcirc$, then

i. There is a 1–1 function $f_v: r_1(l_1) \rightarrow r_2(l_2)$ such that for all $l \in r_1(l_1)$, $l \preceq f(l)$, i.e., everything that is in $r_1(l_1)$ is also in $r_2(l_2)$, possibly with more internal duplication.

ii. For every $l \in r_2(l_2)$, there is an $l' \in r_1(l_1)$ such that $l' \preceq l$, i.e., everything in $r_2(l_2)$ is a copy, possibly with some more internal duplication, of something in $r_1(l_1)$.

Provided that the query does not add cycles to the database, this definition corresponds to the intuition we described above. The problem with cyclic queries is that since the definition is recursive, l-values of nodes that are in a cycle added by the query will never dominate one another, and we shall end up with no way to compare the instances that we get. As we are going to forbid cycles in the query anyway, for other reasons, we shall not discuss here how to modify the definitions to handle cyclic queries.

The next step is to define a relation $\mathbf{I}_1 \preceq \mathbf{I}_2$ between instances. Intuitively, $\mathbf{I}_1 \preceq \mathbf{I}_2$ will mean that $\mathbf{I}_1$ and $\mathbf{I}_2$ contain the same data, but $\mathbf{I}_2$ may have more duplication. This means that $\mathbf{I}_2$ may have more copies of things in $\mathbf{I}_1$, and these copies may have more internal duplication.

Definition 48: Let $\mathbf{I}_1 = \langle I_1, r_1 \rangle$ and $\mathbf{I}_2 = \langle I_2, r_2 \rangle$ be two extensions of $\mathbf{I}$ to $\mathbf{S_Q}$. We say that $\mathbf{I}_1 \preceq \mathbf{I}_2$ iff for each query node v

1. There is a 1–1 function $f_v: I_1(v) \rightarrow I_2(v)$ such that for each $l \in I_1(v)$, $l \preceq f_v(l)$, i.e., everything in $I_1(v)$ is in $I_2(v)$, possibly with more internal duplication.
2. For every $l \in I_2(v)$, there is an $l' \in I_1(v)$ such that $l' \preceq l$.

Definition 49: An extension $\mathbf{I_Q}$ of $\mathbf{I}$ to $\mathbf{S_Q}$ is called a *minimal result of* $\mathbf{Q}$ iff

1. $\mathbf{I_Q}$ is a result of the query, i.e., $\models_{\mathbf{I_Q}} \phi_{\mathbf{Q}}$.
2. $\mathbf{I_Q}$ is minimal, i.e., if $\mathbf{I^*_Q}$ is another extension of $\mathbf{I}$ to $\mathbf{S_Q}$ such that $\mathbf{I^*_Q} \preceq \mathbf{I_Q}$, but $\mathbf{I^*_Q}$ is not isomorphic to $\mathbf{I_Q}$ relative to $\mathbf{S}$, then $\mathbf{I^*_Q}$ is not a result of $\mathbf{Q}$, i.e., $\not\models_{\mathbf{I^*_Q}} \phi_{\mathbf{Q}}$.

Definition 50: A query $\mathbf{Q}$ is called *safe up to duplication on* $\mathbf{I}$ iff $\mathbf{Q}$ has a unique minimal result on $\mathbf{I}$. $\mathbf{Q}$ is *safe up to duplication* iff it is safe on all instances $\mathbf{I}$ of $\mathbf{S}$.

Example 24: If we write the query of Example 23 as $(\mathbf{S_Q}, \phi_1 \wedge \phi_3 \wedge \phi_4)$ we get a somewhat simpler query. This query is safe up to duplication and has the desired result.

A.3. Absolute Safety

The other way to simplify the user queries is to enable the user to avoid having to specify ϕ_3 explicitly. ϕ_3 just says that anything that is allowed (by ϕ_1) to be in the result actually appears in it. In other words what we want to do is to maximize the data in the result. We also want to combine this with minimizing the duplication as above. An *absolutely safe query* will be one that has a unique result under this combined approach, i.e., maximize data- and minimize duplication.

We first define what it means to say that an instance contains more data than another instance.

Definition 51: Let $\mathbf{I}_1 = \langle I_1, r_1 \rangle$ and $\mathbf{I}_2 = \langle I_2, r_2 \rangle$ be two extensions of $\mathbf{I}$ to $\mathbf{S_Q}$. We say that $\mathbf{I}_2$ *contains at least as much data as* $\mathbf{I}_1$, and write $\mathbf{I}_1 \leq \mathbf{I}_2$, iff for each query node v and each element l_1 of $I_1(v)$, there is an element l_2 of $I_2(v)$ that contains the same information, possibly with more internal duplication, i.e., $l_1 \preceq l_2$.

Definition 52: An extension $\mathbf{I_Q}$ of $\mathbf{I}$ to $\mathbf{S_Q}$ *contains the maximum data satisfying* $\mathbf{Q}$ iff

1. $\mathbf{I_Q}$ is a result of $\mathbf{Q}$, i.e., $\models_{\mathbf{I_Q}} \phi_{\mathbf{Q}}$.
2. $\mathbf{I_Q}$ is a maximum result, i.e., if $\mathbf{I^*_Q}$ is an extension of $\mathbf{I}$ to $\mathbf{S_Q}$ that satisfies $\models_{\mathbf{I^*_Q}} \phi_{\mathbf{Q}}$, then $\mathbf{I^*_Q} \leq \mathbf{I_Q}$.

Definition 53: The *absolute result of* $\mathbf{Q}$ is an extension $\mathbf{I_Q}$ of $\mathbf{I}$ to $\mathbf{S_Q}$ such that $\mathbf{I_Q}$ is minimal under $\preceq$ in the class of maximum results, i.e., the class

$$\{\mathbf{I^*_Q} \mid \mathbf{I^*_Q} \text{ contains the maximum data satisfying } \mathbf{Q}\}$$

Definition 54: $\mathbf{Q}$ is *absolutely safe* on $\mathbf{I}$ iff it has a unique absolute result, up to isomorphism. $\mathbf{Q}$ is *absolutely safe* iff it is absolutely safe on all database instances.

Example 25: The query of Example 23 can be written as the absolutely safe query $(\mathbf{S_Q}, \phi_1 \wedge \phi_4)$.

A.4. Undecidability

What we have shown so far is how we can reduce the amount of work the user has to do in order to write a safe query. The language we get is close in this respect to the relational tuple calculus. In order to do this, however, we had to make the definition of what the result of a query is much more complicated and less intuitive.

Besides this, it turns out that all three of the approaches we described are too powerful. We look now at the question how do we test if a given query is safe, either up to isomorphism, up to duplication or absolutely. It is not hard to see that we can reduce testing whether a query in the relational model is safe to testing safety under any of these definitions. Since testing a relational query for safety is undecidable [Pao69] the undecidability of testing for our types of safety follows immediately. In the relational model this undecidability is not a problem. The reason for this is that if we are given a database instance we can test whether the query is safe and we can compute the result when it is. Furthermore, we can give restrictions on the query language that allow the user to write only safe queries, and if all the domains are finite then all relational queries are safe. What is undecidable is just to test whether a query is safe for *all* possible database instances. Our three definitions of safety, on the other hand, are too powerful, since even if we are given a database instance, it is still undecidable whether a query is safe on it.

Theorem 46: There is an acyclic schema $\mathbf{S}$[1], an instance $\mathbf{I}$ of $\mathbf{S}$ and a query $\mathbf{Q}$ on $\mathbf{S}$, such that it is undecidable whether the query is safe on $\mathbf{I}$ up to isomorphism, up to duplication or absolutely.

Proof: We reduce testing the three kinds of safety to testing whether a sentence in a first-order theory with equality and one ternary relation symbol $R(x, y, z)$ has a finite model [Tra50]. The database schema $\mathbf{S}$ will be the empty schema ($V = \emptyset$), which immediately turns both testing for safety on a fixed instance, and on all instances, into the same problem. The query schema $\mathbf{S_Q}$ is shown in Fig. 64. It has $v \prec_{\mathbf{Q}} u$.

[1] The reason for mentioning the fact that it is acyclic is that otherwise the cyclicity of $\mathbf{S}$ might appear to be what causes the undecidability

Figure 64: Undecidable query

Let ϕ be a sentence in the first-order theory. We convert this into an LDM sentence $L(\phi)$ as follows.

1. Introduce a variable x_v for each variable x in ϕ.
2. Replace each quantifier Qx by Qx_v.
3. Replace each atomic formula $x = y$ in ϕ by the LDM formula $x_v =_l y_v$.
4. Replace each atomic formula $R(x, y, z)$ by the LDM formula

$$\phi_R = (\exists w_u)(w_u =_r (x_v, y_v, z_v))$$

The query $\mathbf{Q} = (\mathbf{S_Q}, \psi)$ has ψ equal to

$$L(\phi) \wedge (\forall x_v^1)(\forall x_v^2)(x_v^1 =_r x_v^2 \Rightarrow x_v^1 =_l x_v^2) \wedge (\forall y_u^1)(\forall y_u^2)(y_u^1 =_r y_u^2 \Rightarrow y_u^1 =_l y_u^2)$$
$$\wedge(\forall x_v)(\exists y_u)(x_v\ \pi_1\ y_u \vee x_v\ \pi_2\ y_u \vee x_v\ \pi_3\ y_u)$$

The intention is that this formula says that the result of the query corresponds to a model of ϕ. The three final conjuncts say that the result has no duplication, and that there are no unnecessary elements in v.

We first show that ϕ has a finite model if and only if there is a (finite) instance of $\mathbf{S_Q}$ that satisfies ψ. Let $\mathbf{I}$ be such an instance. We define a finite model $\mathbf{M}$ of ϕ as follows.

The domain of the model is the set of data elements in the instance, i.e., $D_{\mathbf{M}} = \{d \in D \mid (\exists l \in I(v))(r(l) = d)\}$. If a, b and c are in the domain, then (a, b, c) is in $R_{\mathbf{M}}$ if, intuitively, (a, b, c) is in the instance. Formally, this means that there are l-values l_1, l_2 and l_3 in $I(v)$ and l in $I(u)$ such that $r(l) = (l_1, l_2, l_3)$, $l_1 =_r a$, $l_2 =_r b$ and $l_3 =_r c$.

We show that $\mathbf{M}$ is a model of ϕ by induction on the size of ϕ. Let $\overline{\phi}$ be a subformula of ϕ with the free variables $x_1, \ldots, x_n$. Then the free variables of $L(\overline{\phi})$ are $x_v^1, \ldots, x_v^n$. For any assignment of domain elements $a_1, \ldots, a_n$ to these variables, there are unique l-values $l_1, \ldots, l_n$ in $I(v)$ with $r(l_i) = a_i$ for all i, $1 \leq i \leq n$, and there is a unique l in $I(u)$ such that $r(l) = (l_1, \ldots, l_n)$. We now show that

$$\models_{\mathbf{M}} \overline{\phi}(a_1, \ldots, a_n) \Leftrightarrow \models_{\mathbf{I}} L(\overline{\phi})(l_1, \ldots, l_n)$$

For atomic formulas $\overline{\phi}$ of the form $x = y$ this is obvious. For atomic formulas of the form $R(x, y, z)$, $L(\overline{\phi})$ is defined as $(\exists w_u)(w_u =_r (x_v, y_v, z_v))$, and then

$$\begin{aligned} \models_{\mathbf{M}} \overline{\phi}(a_1, a_2, a_3) &\Leftrightarrow (a_1, a_2, a_3) \in R_{\mathbf{M}} \\ &\Leftrightarrow \text{For some } l \text{ in } I(u),\ \ r(l) = (l_1, l_2, l_3) \\ &\Leftrightarrow \models_{\mathbf{I}} L(\overline{\phi})(l_1, l_2, l_3) \end{aligned}$$

The result now follows by a straightforward induction, and shows that $\models_{\mathbf{I}} L(\phi)$ is equivalent to $\models_{\mathbf{M}} \phi$. Therefore $\mathbf{M}$ is a model of ϕ.

For the converse, let $\mathbf{M}$ be a finite model of ϕ. We define an instance $\mathbf{I}$ of $\mathbf{S_Q}$ that satisfies $L(\phi)$ as follows. Let the domain of $\mathbf{M}$ be the finite set A. Introduce new l-values as necessary, and define

$$\begin{array}{ll} I(v) = \{l_a \mid a \in A\}, & r(l_a) = a \\ I(u) = \{l_{R(a,b,c)} \mid a,b,c \in A \text{ and } \models_{\mathbf{M}} R(a,b,c)\}, & r(l_{R(a,b,c)}) = (l_a, l_b, l_c) \end{array}$$

By a straightforward induction, we can show that $\models_{\mathbf{I}} L(\phi)$ holds. It is easy to see that the remaining conjuncts in the definition of ψ also hold, and therefore $\models_{\mathbf{I}} \psi$.

We now return to the undecidability of testing whether a query is safe. Assume that one the three types of safety is decidable, and let ϕ be a sentence over the above first-order logic. We show how to use the test for safety to test whether ϕ has a finite model. Define $\mathbf{Q}$ as above, and apply the decision procedure for the relevant type of safety to $\mathbf{Q}$. If the query is safe, then ϕ has a finite model. If the query is unsafe, however, this does not necessarily mean that ϕ has no finite model. In fact, there are two possibilities

1. $\mathbf{Q}$ has no result.
2. $\mathbf{Q}$ has more than one result.

To distinguish between these two possibilities, and from that to deduce whether ϕ has a finite model, we define a new query $\overline{\mathbf{Q}} = (\mathbf{S_Q}, \overline{\psi})$ in which

$$\overline{\psi} = \psi \vee (\forall x_v^1)(x_v^1 \neq_l x_v^1)$$

Then $\models_{\mathbf{I}} \overline{\psi}$ if and only if $\models_{\mathbf{I}} \psi$ or $\models_{\mathbf{I}} (\forall x_v^1)(x_v^1 \neq_l x_v^1)$. The latter formula is satisfied only by the empty instance $\mathbf{I}_\emptyset$ with $I_\emptyset(u) = I_\emptyset(v) = \emptyset$. Since $\models_{I_\emptyset} \overline{\psi}$, there is always at least one instance that satisfies ψ. Furthermore, it is easy to see that $\mathbf{I}_\emptyset$ is a minimal instance satisfying ψ.

Apply the test for safety to $\overline{\mathbf{Q}}$. We distinguish between the three types of safety, as follows.

1. Safety up to isomorphism. First, assume that $\overline{\mathbf{Q}}$ is unsafe. Since $\mathbf{I}_\emptyset$ satisfies $\overline{\psi}$, the unsafety implies that there is some other instance $\mathbf{I}$ that satisfies $\overline{\psi}$. But then $\mathbf{I}$ satisfies ψ, thus showing that ϕ has a finite model.

   Now assume that $\overline{\mathbf{Q}}$ is safe. Then $\mathbf{I}_\emptyset$ is the only instance satisfying $\overline{\psi}$. Since either zero or more than one instances satisfy ψ, there cannot be any instance satisfying ψ and therefore ϕ does not have a finite model.

2. Safety up to duplication. First, assume that $\overline{\mathbf{Q}}$ is unsafe. Since $\mathbf{I}_\emptyset$ is a minimal instance that satisfies $\overline{\psi}$, the unsafety implies that there is some other minimal instance $\mathbf{I}$ satisfying $\overline{\psi}$. But then $\mathbf{I}$ also satisfies ψ and ϕ has a finite model.

   Now assume that $\overline{\mathbf{Q}}$ is safe. Then $\mathbf{I}_\emptyset$ is the only minimal instance satisfying $\overline{\psi}$. Since there are either zero or more than one minimal instances satisfying ψ, there cannot be any minimal instance satisfying ψ. If there were an instance $\mathbf{I}$ that satisfied ψ, the definition of ψ would imply that $\mathbf{I}$ contained no duplication, and therefore that it must be a minimal instance satisfying ψ. This shows that no instance $\mathbf{I}$ can satisfy ψ, and therefore ϕ does not have a finite model.

3. Absolute safety. First, assume that $\overline{\mathbf{Q}}$ is unsafe. There are two possibilities

   (a) There is no maximum instance satisfying $\overline{\psi}$. On the other hand, we know that $\mathbf{I}_\emptyset$ satisfies $\overline{\psi}$. Since it is not maximum, there must be some other instance satisfying $\overline{\psi}$ and containing at least as much data as $\mathbf{I}_\emptyset$. Such an instance must satisfy ψ and therefore ϕ has a finite model.

(b) There is more than one maximum instance satisfying $\overline{\psi}$. In this case clearly ϕ has a finite model.

Now assume that $\overline{\mathbf{Q}}$ is safe. If the maximum instance that satisfied $\overline{\psi}$ is some instance $\mathbf{I}$ other than $\mathbf{I}_\emptyset$, then it is also a maximum instance satisfying ψ. This implies that $\mathbf{Q}$ is safe, a contradiction. Therefore $\mathbf{I}_\emptyset$ is the only maximum instance satisfying $\overline{\psi}$. If some other instance $\mathbf{I}$ satisfied ψ, the maximality of $\mathbf{I}_\emptyset$ would imply that $\mathbf{I} \leq \mathbf{I}_\emptyset$, a contradiction. Therefore there is no instance satisfying ψ, which shows that ϕ does not have a finite model. ∎

In short, our query language is too powerful. One way to see what is wrong with it is to restrict the schemas to those that correspond to relations. We then get a language that is more powerful than the relational calculus. Essentially, this language has queries whose result is defined implicitly rather than explicitly. For arbitrary first-order structures, Beth's Theorem [CK73] says that for any implicit definition there is an equivalent explicit one, but the theorem does not hold for finite structures which are what we are interested in. Making use of open rather than closed formulas, as we did in the LDM query language, seems therefore to be the way to proceed.

There is in fact a close relation between the LDM query language and our absolutely safe queries. At first it may seem that the LDM query language is actually a special case of the absolutely safe queries. Given a query $\mathbf{Q} = \left\langle \mathbf{S}_\mathbf{Q}, \Phi_\mathbf{Q} \right\rangle$, if we define $\phi = \bigwedge_{v \in V} (\forall x_v) \phi_v(x_v)$ we appear to get an equivalent absolutely safe query. This turns out, however, not to be the case. If this new query is absolutely safe we can indeed show that the results of both queries are the same. The following example, however, shows that even if the original query is safe the new one need not be absolutely safe.

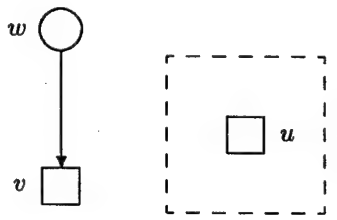


Figure 65: Database schema and logical query

Example 26: Fig. 65 shows the database schema (the node u) and a query on it. The formulas of $\mathbf{Q}$ are

$$\phi_v(x_v) = (\exists x_u)(x_u =_r x_v)$$

i.e., v is a copy of u, and

$$\phi_w(x_w) = (\forall x_v)(x_v \in x_w) \wedge (\forall x_v^1)(\forall x_v^2)(x_v^1 =_l x_v^2)$$

i.e., if there is exactly one l-value in v, collect it into a set in w, otherwise the result at w is empty. If we then define

$$\phi = (\forall x_v)\phi_v(x_v) \wedge (\forall x_w)\phi_w(x_w)$$

we get an query that is not absolutely safe. The reason for this is that if $I(u)$ has more than one element, then both the instance with v containing a copy of u and w being empty, and the instance with any single element of u in v and w collecting it into a set, are incomparable instances that satisfy ϕ.

The reason for the difference between the absolutely safe queries and the LDM query language is that the absolutely safe queries try to *globally* maximize the data in the result whereas in the LDM query language we maximize the data in the nodes one at a time, in a fixed order.

Appendix B

An Alternative Logical Data Model

B.1. The Model

In this appendix we describe an earlier attempt that we made to define a logical data model. We wanted a model that would not allow implicit pointers. If, for example, in an LDM schema, we had two different tuples that contained the same l-value among their components, we would implicitly have a pointer structure. We would be using the l-values as objects having an independent meaning, and this did not seem to us be desirable except when it was explicitly mentioned as part of the schema definition. We therefore tried to define the schema in such a way that such pointers would only be allowed when they were explicitly represented in the schema. We shall describe this approach, and shall see that we get into serious difficulties when we try to define a query language. The logical query language will turn out to require more general constraints on the result than in the LDM model, and this will make it harder to evaluate a query by a bottom-up node-by-node approach. To make a query unambiguous, we will have to make quite complicated restrictions on the form queries can take. As a result, the query language we get is less intuitive than the LDM query language, and in addition, we were not able to find an equivalent algebra. All the same, this approach is instructive, as it illustrates the kind of problems that we encounter when we try to have general constraints on the result of a query.

We shall call the model that we describe in this appendix the "LDM" model, to distinguish it from the real LDM model. We shall not present all the definitions here, but shall give the details mainly where they differ from the LDM model.

Definition 55: An "LDM" schema **S** is a directed *forest* with types associated with the nodes. Cycles will be represented through a new type of node— a pointer node. A leaf in an "LDM" schema is of one of the following types

1. Basic type, written □ (the same as in the LDM model).
2. Pointer type, i.e., the type of v is some other node of **S**. These nodes will be drawn as • together with an arrow to the node that they point to.

Other nodes are of the types ⬭ and ○. To keep the model simple, we leave out the type △

l-values and r-values are defined as in the LDM model, with one additional restriction. We require that no l-value appears as the member of more than one set or as the component of more than one tuple. On the other hand, it can occur any number of times as the r-value of a pointer node.

Definition 56: We shall use the symbol $L(\mathbf{I})$ to represent the set of l-values used in the instance $\mathbf{I}$, i.e., $L(\mathbf{I})$ will be $\cup_{v \in V} I(v)$.

Since the underlying graph of the schema $\mathbf{S}$ is a forest, given a set L_0 of l-values that are used in the instance $\mathbf{I}$, we can define the set of the descendants of these l-values, something we shall need later on.

Definition 57: Let L_0 be a subset of $L(\mathbf{I})$, the set of l-values used in $\mathbf{I}$. $\text{Desc}(L_0) = \cup_{l \in L_0} \text{Desc}(l)$, where $\text{Desc}(l)$, the set of descendants of the l-value $l \in I(v)$, is defined as follows.

1. If v is a leaf, then $\text{Desc}(l) = \{l\}$. Note that we do not follow pointers, and therefore the recursive definition will always terminate.
2. If $\mu(v) = \frown$ and $r(l) = (l_1, \ldots, l_n)$ then $\text{Desc}(l) = \{l\} \cup \text{Desc}(l_1) \cup \cdots \cup \text{Desc}(l_n)$.
3. If $\mu(v) = \bigcirc$, then $\text{Desc}(l) = \{l\} \cup \bigcup_{l' \in r(l)} \text{Desc}(l')$.

We define the "LDM" logic in a similar way to the LDM logic. If a node u points to a node v, the atomic formula $x_v \ \rho \ y_u$ will mean that the value of x_v is what the value of y_u points to. Isomorphism is also defined in a similar way to the LDM model, and we can then show that satisfaction is preserved under isomorphism.

B.2. The Query Language

Defining a logical query language is harder than in the LDM model. The problem is that if we try to do a bottom-up node-by-node evaluation, we put only one copy of each item in each node—no duplication is allowed. However, when we get to, say, a node of type $\frown$, we have a problem. Since no two tuples can contain the same l-value as a component, we may need more than one copy of an object for use at a later node. Furthermore, we have no idea in advance how many copies are needed until we get to that node. This suggests that we have to use some global formula, rather than one formula per node. In general this results in the same problems we had when we tried to define LDM queries this way. However, in this case, we were able to find a restricted class of queries which we were able to handle.

This class of queries has schemas that consist of a single tree with root r and without pointers. We could allow pointers to database nodes, but decided not to, in order to keep the model as simple as possible. The query will also have an "LDM" formula $\phi(x_r)$ that describes what objects should be at the root of the tree. The instances of internal nodes in the tree, unlike those in the LDM model, have no independent meaning. They contain only those objects needed to structure the objects at the root r.

The bound variables in $\phi(x_r)$ range over database nodes and over descendants of r. This turns out to lead to the same problems of implicit definition of the result that we encountered in our first attempt at defining the LDM query language. As we are interested only in the objects at the root, and we want to create other objects only when necessary, we restrict the query language to allow us to refer only to elements of internal nodes of the query that are descendants of the object represented by x_r. We do this by restricting the quantifiers that are allowed in $\phi(x_r)$, in the following way.

1. Let v be a query node whose parent is a node u of type $\bigcirc$. To understand the motivation behind the definition, assume that somehow we have reached the value of the variable x_u from the root. Instead of allowing unrestricted quantification over v, we allow quantification only over those elements of v that are elements of x_u. We write this as $(\forall y_v \in x_u)\psi$. Formally this will be equivalent to $(\forall y_v)(y_v \in x_u \Rightarrow \psi)$.
2. If v is a query node with parent u of type $\frown$ and v is u's k^{th} child, we allow quantification over v only through using quantifiers of the form $(\forall y_v \ \pi_k \ x_u)\psi$. Note that in this case, unlike in the previous

one, the value of x_u uniquely determines the value of y_v. The only reason for using quantification is to avoid having to introduce some other function symbol.

3. Variables can range freely over database nodes, i.e., there are no restrictions on how we may quantify these variables.

The definition of a query is:

Definition 58: A query **Q** on **S** consists of a pair $\langle \mathbf{S}', \phi \rangle$ where

1. $\mathbf{S}'$, the *query schema*, is a schema with no pointer nodes, in which the underlying graph is a tree with root r.
2. $\phi(x_r)$ is an "LDM" formula with the properties:
   (a) $\phi(x_r)$ has exactly one free variable and this variable is of sort r.
   (b) Every quantifier in ϕ on a variable that ranges over a query node is either of the form $(\forall y_w \in z_u)\psi$ or $(\forall y_w\ \pi_k\ z_u)\psi$, where z_u is a variable that occurs free in the subformula ψ.

If **Q** is a query on **S**, $\mathbf{S_Q}$ will denote the final schema, i.e., **S** combined with $\mathbf{S}'$.

When evaluating a query, we will have to put duplicates in some of the nodes. We first define precisely what duplication is, by defining an equivalence relation between l-values in two extensions $\mathbf{I}_1$ and $\mathbf{I}_2$ of **I** to $\mathbf{S_Q}$. Two l-values in the database instance **I** will be equivalent only when they are equal. On the other hand, two l-values in query nodes will be equivalent provided that when we follow all the paths from these l-values down to the leaves we get the same information.

Definition 59: Let l_1 be an element of $I_1(v)$ and l_2 an element of $I_2(v)$. We say that l_1 and l_2 are *equivalent*, and write $l_1 \equiv l_2$, if the following holds.

1. If v is a node in the database schema **S**, then $l_1 = l_2$.
2. If v is a node in the query schema $\mathbf{S}'$, then
   (a) If v is a leaf of any type, then $r_1(l_1) = r_2(l_2)$.
   (b) If v is of type (⬭) then all the components are equivalent, i.e., for each i, $1 \le i \le n$, $\Pi_i(r_1(l_1)) \equiv \Pi_i(r_2(l_2))$.
   (c) If v is of type ○, then for each $l \in r_1(l_1)$, there is an $l' \in r_2(l_2)$ such that $l \equiv l'$, and vice versa. Note that this allows duplication *inside* sets. This will be specifically forbidden in the definition of the result of a query.

We shall now give a list of properties that we would like the result of a query to satisfy. These properties are similar to those that the result of an LDM query satisfies, as in Lemma 22. We shall use these properties as the *definition* of the result, and then investigate when it is well-defined.

Definition 60: The result of **Q** on an instance **I** of **S** is an extension $\mathbf{I_Q}$ of **I** to $\mathbf{S_Q}$ that satisfies:

1. For every l in $I_{\mathbf{Q}}(r)$, $\models_{\mathbf{I_Q}} \phi_r(l)$.
2. There is no duplication at the root, i.e., if l_1 and l_2 are l-values in $I_{\mathbf{Q}}(r)$ and $l_1 \equiv l_2$, then $l_1 = l_2$.
3. There are no unnecessary l-values in the result, so that whenever an l-value l is used in the result, it must be a descendant of some l-value in $I_{\mathbf{Q}}(r)$, i.e., it must be in $\mathrm{Desc}(I_{\mathbf{Q}}(r))$.

4. There is no duplication in nodes of type O, i.e., if v is a query node of type O and l is an element of $I_{\mathbf{Q}}(v)$, then $l_1, l_2 \in r(l)$ together with $l_1 \equiv l_2$ imply that $l_1 = l_2$.
5. The result is maximal, i.e., if $\mathbf{I}^*_{\mathbf{Q}}$ is another extension of $\mathbf{I}$ to $\mathbf{S_Q}$ that also satisfies 1–4, then $\mathbf{I}^*_{\mathbf{Q}}$ can be embedded in an instance isomorphic to $\mathbf{I_Q}$.

Let $\mathbf{I_Q}$ be an extension of $\mathbf{I}$ to $\mathbf{Q}$ and let L_0 be a set of l-values used in $\mathbf{I_Q}$. We shall define Restrict($\mathbf{I}, L_0$) as the minimal extension of $\mathbf{I}$ to $\mathbf{S_Q}$ that uses all the l-values in L_0.

Definition 61: Let $\mathbf{I_Q}$ be an extension of $\mathbf{I}$ to $\mathbf{S_Q}$. Let L_0 be a set of l-values that are used in the result of the query (but not in the database). The *restriction* of $\mathbf{I_Q}$ to L_0, written Restrict($\mathbf{I}, L_0$) = $\mathbf{I}^*_{\mathbf{Q}}$, is the following instance of $\mathbf{S_Q}$.

1. For each query node w, $I^*_{\mathbf{Q}}(w)$ is equal to $I(w) \cap \mathrm{Desc}(L_0)$.
2. For each database node w, $I^*_{\mathbf{Q}}(w)$ is equal to $I(w)$.

Lemma 47: Restrict($\mathbf{I}, L_0$) is an instance of $\mathbf{S_Q}$. ∎

We next show that because of the restrictions on the form $\phi(x_r)$ can have, we are able to test if an object should be in the result of the query, by looking only at the descendants of this object, and not at anything else in any query node.

Lemma 48: Let $\mathbf{I_Q}$ be an extension of $\mathbf{I}$ to $\mathbf{Q}$, and let l be an l-value in $I_{\mathbf{Q}}(r)$. Then

$$\models_{\mathbf{I_Q}} \phi(l) \Leftrightarrow \models_{\mathrm{Restrict}(\mathbf{I_Q},\{l\})} \phi(l)$$

Proof: Let $\mathbf{I}^*_{\mathbf{Q}} = \mathrm{Restrict}(\mathbf{I_Q}, \{l\})$. The result will follow immediately from the following inductive assertion, by taking $\psi = \phi(x_r)$.

Let $\psi(x^1_{v_1}, \ldots, x^n_{v_n})$ be a subformula of ϕ with free variables $x^1_{v_1}, \ldots, x^n_{v_n}$. Let $l_i \in I^*_{\mathbf{Q}}(v_i)$, $i = 1, \ldots, n$. Then

$$\models_{\mathbf{I_Q}} \psi(l_1, \ldots, l_n) \Leftrightarrow \models_{\mathbf{I}^*_{\mathbf{Q}}} \psi(l_1, \ldots, l_n)$$

This is trivial whenever ψ is an atomic formula or is of the form $\neg\psi'$ or $\psi_1 \vee \psi_2$. When we quantify over a variable whose sort is a database node, the result is also immediate. The remaining cases are

1. ψ is $(\forall y_w\ \pi_t\ x_{v_i})\psi'(x^1_{v_1}, \ldots, x^n_{v_n}, y_w)$. By the definition of the restricted quantification,

$$\models_{\mathbf{I}^*_{\mathbf{Q}}} (\forall y_w\ \pi_t\ x_{v_i})\psi(l_1, \ldots, l_n)$$

is equivalent to

$$\models_{\mathbf{I}^*_{\mathbf{Q}}} \psi'(l_1, \ldots, l_n, \Pi_t(l_i))$$

Since $l_i \in \mathbf{I}^*_{\mathbf{Q}}(v_i)$, $\Pi_t(l_i)$ is in $\mathbf{I}^*_{\mathbf{Q}}(w)$, and the inductive hypothesis implies that this is equivalent to $\models_{\mathbf{I_Q}} \psi'(l_1, \ldots, l_n, \Pi_t(l_i))$, and therefore to

$$\models_{\mathbf{I_Q}} (\forall y_w\ \pi_t\ x_{v_i})\psi'(l_1, \ldots, l_n)$$

2. ψ is $(\forall y_w \in x_{v_i})\psi'(x^1_{v_1}, \ldots, x^n_{v_n}, y_w)$. By definition,

$$\models_{\mathbf{I}^*_\mathbf{Q}} (\forall y_w \in x_{v_i})\psi(l_1, \ldots, l_n)$$

is equivalent to $\models_{\mathbf{I}^*_\mathbf{Q}} \psi'(l_1, \ldots, l_n, l)$, for all l in $r(l_i)$. Since l_i is an element of $I^*_\mathbf{Q}(v_i)$, all the members of $r(l_i)$ are in $I^*_\mathbf{Q}(w)$, and therefore, by the inductive hypothesis, this is equivalent to $\models_{\mathbf{I}_\mathbf{Q}} \psi'(l_i, \ldots, l_n, l)$ for all l in $r(l_i)$, i.e., to $\models_{\mathbf{I}_\mathbf{Q}} \psi(l_1, \ldots, l_n)$. ∎

This lemma is the crucial one behind the definition of the query language. It says that the truth of a formula $\phi(l)$ depends only on the contents of l and its descendants, and not on anything else in the result of the query, and therefore we can look for the objects we want to put at the root, one at a time, without considering any interactions between these objects.

Theorem 49: Let $\mathbf{I}^1_\mathbf{Q}$, $\mathbf{I}^2_\mathbf{Q}$ be two results of $\mathbf{Q}$. Then $\mathbf{I}^1_\mathbf{Q}$ and $\mathbf{I}^2_\mathbf{Q}$ are isomorphic relative to $\mathbf{S}$.

Proof: By part 5 of Definition 60, $\mathbf{I}^1_\mathbf{Q}$ can be isomorphically embedded in $\mathbf{I}^2_\mathbf{Q}$, and vice versa. If f and g are these isomorphisms, the fact that all instances are finite implies that f and g are 1–1 and onto, and hence that the instances $\mathbf{I}^1_\mathbf{Q}$ and $\mathbf{I}^2_\mathbf{Q}$ are isomorphic. ∎

B.3. Safety

In the previous section we assumed that the query $\mathbf{Q}$ had a result, and showed that then the result is unique. If we were to remove the requirement that an instance be finite, we would expect a result always to exist, as is shown by the following informal argument.

Define an extension $\mathbf{I}_\mathbf{Q}$ of $\mathbf{I}$ to $\mathbf{Q}$ by defining $I_\mathbf{Q}(v)$ at each leaf v to be an infinite set of l-values, one for each possible data element $d \in D$. Going up the tree, put all tuples made out of the children of a node of type ⌓ at that node. Each node of type ○ contains the entire powerset of its child. In both cases, no l-value can be in more than one tuple or in more than one set, so we have to create duplicate l-values when necessary. We repeat this until reaching the root r, and then remove all the l-values in $I_\mathbf{Q}(r)$ that do not satisfy ϕ. Finally, we restrict $\mathbf{I}_\mathbf{Q}$ to the descendants of the l-values in $I_\mathbf{Q}(r)$. It turns out that whenever this instance is finite, it is the result of the query. We can also show the converse, that whenever the query has a result, it is isomorphic to the "instance" we constructed here, and hence this "instance" is finite. As in the relational model, we define

Definition 62: $\mathbf{Q}$ is safe on an instance $\mathbf{I}$ of $\mathbf{S}$ iff $\mathbf{Q}$ has a result on $\mathbf{I}$.

This definition of safety turns out to be closely related to the safe queries in the relational calculus and in the LDM model, as it captures a similar finiteness property. We now formalize the construction that we described above. We first have to define duplication of l-values more precisely.

We do this by defining a function Dup that has two arguments—an instance $\mathbf{I}_0$ of the database together with some of the query nodes, and a set L_0 of l-values in $L(\mathbf{I}_0)$. L_0 is the set of l-values that we want to duplicate. The result of Dup consists of:

1. An instance $\mathbf{I}_1$ that is a superset of $\mathbf{I}_0$.

2. A function Copy that maps the duplicated l-values in $\text{Desc}(L_0)$ into their duplicates.

Definition 63: Let $\mathbf{I}_0$ be an instance of the database together with some of the query nodes, and let L_0 be a set of l-values used in $\mathbf{I}_0$. The result of the function $\mathrm{Dup}(L_0, \mathbf{I}_0)$ consists of a pair $(\mathbf{I}_1, \mathrm{Copy})$. Let $L_1 = \mathrm{Desc}(L_0)$. For each l in L_1, we introduce a new l-value which will be called $\mathrm{Copy}(l)$. For each node v, $I_1(v)$ will be $I_0(v)$ together with the relevant new l-values, i.e., $I_1(v) = I_0(v) \cup \mathrm{Copy}[L_1 \cap I_0(v)]$. For each new l-value $\mathrm{Copy}(l)$, where $l \in L_1$, we now define $r(\mathrm{Copy}(l))$. Let l be an element of $I_0(w)$.

1. If w is a leaf, then $r\big(\mathrm{Copy}(l)\big) = r(l)$.
2. If w is of type $\backsim$, and $r(l) = (l_1, \ldots, l_n)$, then
$$r\Big(\mathrm{Copy}(l)\Big) = \Big(\mathrm{Copy}(l_1), \ldots, \mathrm{Copy}(l_n)\Big)$$
3. If w is of type $\bigcirc$, then $r\big(\mathrm{Copy}(l)\big) = \{\mathrm{Copy}(l') \mid l' \in r(l)\}$.

Lemma 50: Let $\mathrm{Dup}(L_0, \mathbf{I}_0) = (\mathbf{I}_1, \mathrm{Copy})$. Then

1. $\mathbf{I}_1$ is an instance of the schema.
2. The result of Dup is well-defined, i.e., if we choose different new l-values we get an isomorphic result.
3. The domain of the function Copy is the set $\mathrm{Desc}(L_0)$ of descendants of the l-values in L_0, and its range is $L(\mathbf{I}_1) - L(\mathbf{I}_0)$. ∎

We now return to the construction of the result of $\mathbf{Q}$. We are given an instance $\mathbf{I}$ of $\mathbf{S}$, and we construct an extension $\mathbf{I_Q}$ of $\mathbf{I}$ to $\mathbf{S_Q}$.

1. If v is a query leaf, its r-values will be
$$R_0 = \{d_1, \ldots, d_k\} \cup \bigcup_{\substack{\text{w is in a database} \\ \text{node of type } \square}} r[I(w)]$$
where $d_1, \ldots, d_k$ are the elements of D that appear in ϕ. This resembles the safety requirement in the relational model and in the LDM model. $I_{\mathbf{Q}}(v)$ contains one l-value for each element of R_0, with the corresponding r-values.
2. If v is a node of type $(\backsim, n, v_1, \ldots, v_n)$, we would like $I_{\mathbf{Q}}(v)$ to contain all the tuples in $I_{\mathbf{Q}}(v_1) \times \cdots \times I_{\mathbf{Q}}(v_n)$. However, since no l-value can be in more than one tuple, we have to create duplicate l-values. We therefore apply the function $\mathrm{Dup}(I_{\mathbf{Q}}(v_i), \mathbf{I_Q})$ $\Pi_{j \neq i} |I(v_j)| - 1$ times. Then for each tuple $(l_1, \ldots, l_n)$ where each l_i is in the original instance of v_i we introduce a new l-value $l \in I_{\mathbf{Q}}(v)$, whose r-value is a tuple whose components are equivalent to the l_i's. We can do this is such a way that we use each l-value in the instances of the children of v exactly once.
3. If v is of type $\bigcirc$ with child w we do a similar construction, but this time we duplicate $I_{\mathbf{Q}}(w)$ $2^{(|I(w)|-1)} - 1$ times, so that we can put all possible subsets as r-values in $I_{\mathbf{Q}}(v)$ without repeating l-values.

We then define L_0 as those elements l of $I_{\mathbf{Q}}(r)$ that satisfy $\models_{\mathbf{I_Q}} \phi(l)$ and replace $\mathbf{I_Q}$ by $\mathrm{Restrict}(\mathbf{I_Q}, L_0)$.

Lemma 51: Let $\mathbf{Q}$ be a query on a schema $\mathbf{S}$ with instance $\mathbf{I}$. Let $\mathbf{I}^*$ be the instance created by the above construction. Then $\mathbf{Q}$ is safe on $\mathbf{I}$ iff $\mathbf{I}^*$ is the result of $\mathbf{Q}$.

Proof: The first direction, showing that **Q** is safe whenever $\mathbf{I}^*$ is the result of **Q**.

To show the converse, assume that $\mathbf{I}^*$ is not the result of **Q**. It is easy to see that $\mathbf{I}^*$ satisfies parts 1–4 of Definition 60, and therefore must violate the fifth part, i.e., the maximality condition. Therefore, there is some other instance $\mathbf{I}^{**}$ satisfying 1–4 of Definition 60 that cannot be isomorphically embedded in $\mathbf{I}^*$. If all the r-values of the query leaves in $\mathbf{I}^{**}$ are also r-values of the query leaves in $\mathbf{I}^*$, it is not hard to see that since our construction considers all possible combinations of these l-values it must be possible to embed $\mathbf{I}^{**}$ in $\mathbf{I}^*$. Therefore there must be some data element d_0 that is an r-value of some query leaf in $\mathbf{I}^{**}$ but not in $\mathbf{I}^*$. If d_0 were in the set R_0, we would consider all objects involving it when constructing $\mathbf{I}^*$ and therefore $d_0 \notin R_0$. But then, as in the LDM query language, we can replace d_0 by any such constant, i.e., any element of $D - R_0$, and get an object that satisfies ϕ. Therefore **Q** is unsafe. ∎

Bibliography

[A*76] M. M. Astrahan et al. System R: A relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.

[AB84] S. Abiteboul and N. Bidoit. Non first normal form relations to represent hierarchically organized data. In *Proc. Third Annual ACM Symposium on Principles of Database Systems*, pages 191–200, ACM, Waterloo, Ontario, 1984.

[Acz85] P. Aczel. Notes on non-well-founded sets. 1985. Unpublished Manuscript.

[ANS75] ANSI/X3/SPARC. Study group on data base management systems: interim report. *File Description and Translation*, 7(2), 1975.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[BBG78] C. Beeri, P. A. Bernstein, and N. Goodman. A sophisticate's introduction to database normalization theory. In *Proc. Fourth Intl. Conf. on Very Large Data Bases*, pages 113–124, Berlin, 1978.

[Bor78] S. A. Borkin. Data model equivalence. In *Proc. Fourth Intl. Conf. on Very Large Data Bases*, pages 526–534, Berlin, 1978.

[BRR82] A. Balsamini, M. Rafanelli, and F. L. Ricci. *GRASS: A Logical Model for Statistical Databases.* Technical Report TR-39, IASI-CNR, 1982.

[Che76] P. P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[CK73] C. C. Chang and H. J. Keisler. *Model Theory.* North-Holland, Amsterdam, 1973.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[COD71] CODASYL. *CODASYL Data Base Task Group April 71 Report.* ACM, New York, 1971.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Dat80] C. J. Date. An introduction to the unified database language (UDL). In *Proc. Sixth Intl. Conf. on Very Large Data Bases*, pages 15–32, IEEE, Montreal, Quebec, 1980.

[Dat81] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, Reading, Mass., 1981.

[Day79] U. Dayal. *Schema-Mapping Problems in Database Systems.* PhD thesis, Center for Research in Computing Technology, Harvard University, 1979.

[DB82] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, 1982.

[FK77] A. L. Furtado and L. Kerschberg. An algebra of quotient relations. In *Proc. ACM Int'l Conf. on Management of Data*, pages 1–8, ACM, Toronto, Ontario, 1977.

[GDB82] D. Gangopadhyay, U. Dayal, and J. C. Browne. Semantics of network data manipulation languages: an object-oriented approach. In *Proc. Eighth Intl. Conf. on Very Large Data Bases*, IEEE, 1982.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, 1979.

[Gra79] M. H. Graham. NETS: operations and logic. In F. H. Lochovsky, editor, *A Panache of DBMS Ideas II*, pages 152–179, Tech. Report CSRG-101, Computer Systems Research Group, Univeristy of Toronto, 1979.

[Har78] W. T. Hardgrave. *Ambiguity in Processing Boolean Queries on TDMS Tree-Structures: A Study of Four Different Philosophies.* Technical Report IFSM TR-35, University of Maryland, 1978.

[HM81] M. Hammer and D. McLeod. Database description with SDM—A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[HN84] L. J. Henschen and S. A. Naqvi. On compiling queries in recursive first-order databases. *Journal of the ACM*, 31(1):47–85, 1984.

[Hul84] R. Hull. Relative information capacity of simple relational database schemata. In *Proc. Third Annual ACM Symposium on Principles of Database Systems*, pages 97–109, ACM, Waterloo, Ontario, 1984.

[HY82] R. Hull and C. K. Yap. The format model: A theory of database organization. In *Proc. First Annual ACM Symposium on Principles of Database Systems*, pages 205–211, ACM, Los Angeles, CA, 1982.

[IBM78] IBM. *IMS/VS: General Information.* GH20–1260, IBM, White Plains, NY, 1978.

[Jac79] B. E. Jacobs. *Application of Database Logic to Database Design.* Technical Report TR–892, University of Maryland at College Park, 1979.

[Jac80] B. E. Jacobs. *Applications of Database Logic to the View Update Problem.* Technical Report TR–960, University of Maryland at College Park, 1980.

[Jac82] B. E. Jacobs. On database logic. *Journal of the ACM*, 29(2):310–332, 1982.

[JS82] G. Jaeschke and H.-J. Schek. Remarks on the algebra of non first normal form relations. In *Proc. First Annual ACM Symposium on Principles of Database Systems*, pages 124–138, ACM, Los Angeles, CA, 1982.

[Kay75] M. H. Kay. An assessment of the CODASYL DDL for use with a relational subschema. In B. C. M. Douqué and G. M. Nijssen, editors, *Data Base Description*, pages 199–214, North-Holland, Amsterdam, 1975.

[Kob80] I. Kobayashi. *An Overview of the Database Mangement Technology.* Technical Report TRCS–4–1, Sanno College, Kanagawa 259–11, 1980.

[KV84] G. M. Kuper and M. Y. Vardi. A new approach to database logic. In *Proc. Third Annual ACM Symposium on Principles of Database Systems*, pages 86–96, ACM, Waterloo, Ontario, 1984.

[KV85] G. M. Kuper and M. Y. Vardi. On the expressive power of the logical data model. In *Proc. ACM Int'l Conf. on Management of Data*, pages 180–189, ACM, Austin, TX, 1985.

[Mak77] A. Makinouchi. A consideration on normal form of not-necessarily normalized relations in the relational data model. In *Proc. Third Intl. Conf. on Very Large Data Bases*, pages 447–453, IEEE, Tokyo, Japan, 1977.

[MMSU81] D. Maier, A. O. Mendelzon, F. Sadri, and J. D. Ullman. Adequacy of decompositions of relational databases. In H. Gallaire, J. Minker, and J. M. Nicolas, editors, *Advances in Database Theory*, pages 101–114, Plenum Press, 1981.

[MP82] F. Manola and A. Pirotte. CQLF—A query language for CODASYL-type databases. In *Proc. ACM Int'l Conf. on Management of Data*, pages 94–103, ACM, Orlando, FL, 1982.

[NG78] J. M. Nicolas and H. Gallaire. Database: Theory vs. interpretation. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 33–54, Plenum Press, 1978.

[OO84] G. Ozsoyoglu and Z. M. Ozsoyoglu. SSDB—An architecture for statistical databases. In *Proc. Fourth JCIT*, pages 327–341, Jerusalem, 1984.

[OY85] Z. M. Ozsoyoglu and L.-Y. Yuan. A normal form for nested relations. In *Proc. Fourth Annual ACM Symposium on Principles of Database Systems*, pages 251–260, ACM, Portland, OR, 1985.

[Pao69] R. A. Di Paola. The recursive unsolvability of the decision problem for the class of definite formulas. *Journal of the ACM*, 16(2):324–327, April 1969.

[Rei78] R. Reiter. Deductive question answering in relational databases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 147–177, Plenum Press, 1978.

[Rei84] R. Reiter. Towards a logical reconstruction of relational database theory. In M. L. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, pages 191–233, Springer-Verlag, 1984.

[RR83] M. Rafanelli and F. L. Ricci. *A Data Definition Language for a Statistical Database.* Technical Report TR-62, IASI-CNR, July 1983.

[RR84] M. Rafanelli and F. L. Ricci. *STAQUEL: A Query Language for Statistical Databases.* Technical Report TR-96, IASI-CNR, October 1984.

[Sch38] A. Schmidt. Über deduktive Theorien mit mehreren Sorten von Grunddingen. *Math. Ann.*, 115:485–506, 1938.

[SP82] H.-J. Scheck and P. Pistor. Data structures for an integrated data base management and information retrieval system. In *Proc. Fourth Intl. Conf. on Very Large Data Bases*, IEEE, 1982.

[SS75] H. A. Schmid and J. R. Swenson. On the semantics of the relational data model. In *Proc. ACM Int'l Conf. on Management of Data*, pages 211–223, ACM, San Jose, CA, 1975.

[SS77a] J. M. Smith and D. C. P. Smith. Database abstractions: Aggregation. *Communications of the ACM*, 20(6):405–413, 1977.

[SS77b] J. M. Smith and D. C. P. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.

[Sto77] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1977.

[SWKH76] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, 1976.

[Tod76] S. J. P. Todd. The Peterlee Relational Test Vehicle—A system overview. *IBM Systems J.*, 15(4):285–308, 1976.

[Tra50] B. A. Trachtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Dokl. Akad. Nauk SSSR*, 70:569–572, 1950.

[Tsi76] D. C. Tsichritzis. LSL: A link and selector language. In *Proc. ACM Int'l Conf. on Management of Data*, pages 123–133, ACM, Washington, D. C., 1976.

[Ull82] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, Rockville, MD, 1982.

[Ull85] J. D. Ullman. Implementation of logical query languages for databases. In *Proc. ACM Int'l Conf. on Management of Data*, ACM, Austin, TX, 1985.

[Var82] M. Y. Vardi. The complexity of relational query languages. In *Proc. Fourteenth Annual ACM Symposium on the Theory of Computing*, pages 137–146, ACM, San Francisco, CA, 1982.

[Var83] M. Y. Vardi. Review of [Jac82]. *Zentralblatt für Mathematik*, 497.68061, 1983.

[Wie83] G. Wiederhold. *Database Design.* McGraw-Hill, New York, 1983.

[Zlo77] M. M. Zloof. Query-by-Example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.